AD-A196 204

RADC-TR-87-240
[illegible] Report
[illegible]

EVALUATION OF HARDWARE ARCHITECTURES FOR HF ANTENNA ARRAYS

The University of Kansas

[illegible], Victor Frost, Roger Spohn,
[illegible]

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TISL 5481-2	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-240
6a. NAME OF PERFORMING ORGANIZATION The University of Kansas 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (DCCD)
6c. ADDRESS (City, State, and ZIP Code) Telecommunications and Information Sciences Lab 224 Nichols Hall - Campus West Lawrence KS 66045	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) DCCD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0205
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62702F; PROJECT NO. 4519; TASK NO 61; WORK UNIT ACCESSION NO. P4

11. TITLE (Include Security Classification)
EVALUATION OF HARDWARE ARCHITECTURES FOR HF ANTENNA ARRAYS

12. PERSONAL AUTHOR(S)
R. K. Balasubramaniam, Victor Frost, Roger Spohn, Ryan Moates, Stephen Fechtel

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM May 86 TO May 87	January 1988	218

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
25	02		HF Adaptive Arrays
25	05		HF Channel Simulation
			HF Communication Systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A variety of hardware architectures can be used for the implementation of HF adaptive signal processing algorithms. The purpose of this study is to evaluate various hardware structures, enumerate their advantages and disadvantages relative to the HF adaptive array problem, and to recommend the architectures best suited to the adaptive HF array systems. The hardware structures are compared based upon complexity which is determined by issues such as computational bounds, input requirements and characteristics of hardware structures. This evaluation is performed in a hierarchical manner. Application of general microprocessor technology is considered first. Integrated circuits tailored for digital signal processing applications is investigated next. Finally VLSI signal processing technology is considered. Included here are Systolic and Wavefront architectures.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Peter J. Ritchie	22b. TELEPHONE (Include Area Code) (315) 330-3224 22c. OFFICE SYMBOL RADC (DCCD)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

Conventional antenna receiving systems are susceptible to performance degradation due to the presence of undesired noise signals (deliberate or natural) that enter the system. Extensive research has been conducted in the area of adaptive antenna arrays as a means of compensating for the inevitable presence of these interference signals.

Adaptive arrays can provide a vital element of flexibility to a communication system. They can respond to changes in the interference environment by steering nulls and reducing sidelobes in the directions of the interferences, while maintaining an acceptable level of response in the direction of the desired signal. These features make adaptive arrays systems very attractive for applications in which the environment is changing or unknown.

The heart of the adaptive array system is the controlling algorithm. It determines not only the method that is used to adapt, but also the speed of adaptation. The focus of this study is to evaluate the various hardware architectures that can be used for implementation of the adaptive control algorithms.

Three adaptive control algorithms have been considered for hardware evaluation. These algorithms comprise a fairly representative set of adaptive algorithms in general. These algorithms are:

1. Least Mean Square Algorithm
2. Constrained LMS Algorithm

3. Update Covariance Algorithm

The hardware structures considered are the general purpose microprocessor architecture, the digital signal processor architecture and the VLSI architecture. These hardware structure comparisons are based on complexity. The complexity of the hardware structure is determined by considering the computational bounds, input requirements and characteristics of the architecture. The computational bounds of the algorithms studied here is shown in table 1.1.

This study is concerned with the evaluation of a communication system operating in the HF frequency band. The HF band, which spans the 3-30 MHz range, is commonly used in military communication systems, and has been modeled as a slowly varying channel. It has been determined that the HF channel can be considered stationary for times in the order of 100ms. Thus, an adaptive algorithm must complete a convergence (obtain a set of optimum weights) during this period. The average number of iterations required by the adaptive algorithm for a convergence has been determined by previous simulation studies [1]. These previous simulation results guided the input requirement for each algorithm. Since the computational time per iteration is small, fixed-point implementation is considered. The architectural analysis in this study is focussed on 36 antenna elements.

The first section of this report is devoted to the hardware architectures used in the evaluation of the adaptive algorithms. The characteristics of the architectures are

Table 1.1 Complexity of the adaptive algorithms considered for study

Algorithm	Complex Multiply	Complex Adds	Memory
LMS	2N	2N + 1	2N + 1
c-LMS	$N^2 + 2N$	$N^2 + 3N$	$N^2 + 3N$
Update covariance	$3.5N^2 + 4.5\ N$	$2N^2 + 2N$	$N^2 + 2N$

highlighted. The Mc68020 (general purpose microprocessor) and LM32900 (digital signal processor) were chosen for the architectural analysis.

In the following sections the feasibility of the hardware structures for each of the algorithms is examined in detail. Presented is a loading analysis procedure which aids in determining an initial estimate on the computational loading, hence, the complexity of the algorithm. Possible system design architectures are provided for feasible hardware structures. The time/iteration achieved and the memory requirements for the data storage for these system design architectures is determined.

The LMS algorithm was considered first. As the LMS algorithm did not possess any compute bound operations, the VLSI computing structure was not suitable. The DSP architecture was the best suited, for the complexity of the algorithm using DSP architecture was found to be much lower than the complexity using the microprocessor architecture. The c-LMS algorithm was considered next. The complexity in implementation of the algorithm using the microprocessor was too high and therefore not feasible. The Systolic architecture was determined to be better suited than the Wavefront architecture for the compute bound operations involved. Again, in this case the DSP architecture was the architecture of choice. Finally, the update covariance algorithm was considered. The complexity of the Wavefront architecture was high and therefore not feasible. The DSP architecture is once again the recommended

architecture.

The final section presents recommendations on the hardware structures best suited for the adaptive control algorithms.

2.0 HARDWARE ARCHITECTURES

This chapter introduces the various hardware architectures used for implementation of HF adaptive antenna signal processing algorithms. The hardware architectures considered for this study are the general purpose microprocessor architecture, digital signal processor (DSP) architecture and VLSI architecture. The characteristics of these architectures are discussed in this chapter.

2.1 General purpose microprocessor architecture

One of the computations often encountered in signal processing algorithms is the multiply-add. This computation is denoted as AB + C -> C, which implies that the new value of C is obtained by the addition of the old value of C and the product of the two quantities A and B. The general purpose microprocessor architecture's arithmetic logic unit (ALU) is not optimized to this computation. As we are dealing with complex quantities, we are interested in complex multiply and add.

The Mc 68020 microprocessor [2] was chosen for the analysis. The characteristics of this microprocessor are:

1) cycle time of 60ns
2) multiply to add time ratio of 6:1
3) performs a complex multiply-add at the rate of 0.5 MOPS (figure 2.1)

From figure 2.1 it can be seen that the multiply to add time ratio is 7:1. This analysis uses an averaged ratio of 6:1. The

Figure 2.1 Assembly code to perform complex multiply-add computation on the Mc 68020 microprocessor

The computation involves the product of complex operand A (moved in from memory location pointed by address register A1 to data registers D0 and D1) and complex operand B (present in memory location pointed to by address register A0). The result operand AB is present in data registers D2 and D3. The final addition is performed to get the new value of C by peforming AB + C -> C, where the old value of operand C is stored in data registers D6 and D7.

code	cycles (worst case)
move <A1+ > D0	7
move <A1 > D1	7
move D0, D2	3
move D1, D3	3
mul <A0+ > D2	48
mul <A0 > D3	48
sub D3, D2	7
move D0, D3	3
move D1, D4	3
mul <A0 > D3	48
mul <-A0 > D4	49
add D4, D3	7
add D2, D6	7
add D3, D7	7

add operation indicated in figure 2.1 has the destination of the result as the register, but on few occasions we used the add operation in which the destination of the result was a memory location which consumes 10 cycles, providing a multiply to add ratio of 5:1. Thus we averaged the two to obtain the multiply to add time ratio of 6:1.

The number of cycles the microprocessor takes to perform an instruction is indicated for the worst and the best case [2]. For this analysis, the worst case was determined when the microprocessor required 247 cycles to perform the 8 operations to provide a computational rate of 0.5 MOPS (see figure 2.1).

2.2 Digital signal processor (DSP) architecture

The digital signal processor's arithmetic logic unit is optimized to perform the multiply add operation. Every instruction in the digital signal processor takes a fixed number of cycles and does not vary as in microprocessor architecture. Another characteristic of the DSP architecture is that the time taken to perform an addition is on the same order as the time taken to perform a multiplication.

The DSP chosen for the analysis was the LM 32900 [3]. The processor's three-level pipelined architecture allows the overlapping of instruction fetching, decoding and execution, such that all three operations occur in a 100ns cycle. In addition the execution of most instructions is only one cycle long. The pipeline extends to four levels, the fourth level being the accumulation operation for the multiply and add or

multiply and subtract instructions. Consequently, (from [3]) a multiplication-accumulation will likewise take, in effect, only one machine cycle when followed by another. Two operands are fetched, multiplied, and scaled in one machine cycle; the accumulation occurs in the next cycle, but when followed by another multiplication-accumulation it occurs at the same time as the next multiplication. Thus if a sum of products is to be calculated and there are 32 products, the sum of 32 products would be available in 33 cycles (n+1, where n is the number of products). Program is written to perform the complex multiply add computation (see figure 2.2). The result of our analysis show the DSP chip chosen has a computational capacity of 5.7 MOPS and requires 14 cycles to perform the 8 operations for a complex multiply add.

2.3 VLSI architecture

There are three primary reasons why general purpose uniprocessor computers, especially microcomputers, have met with limited success for high-speed signal processing applications [4]. First it has previously been shown [26], [27] that the major computational requirements for many important real-time signal processing tasks can be reduced to a common set of basic matrix operations. These matrix operations involve a variety of elementary operations such as multiplication, vector rotation, and trigonometric functions which general purpose uniprocessors are not efficient in calculating. Secondly, general purpose computer architectures provide only cumbersome address

Figure 2.2 Assembly code to compute a complex multiply-add on the DSP chip LM 32900

```
move    r4 >> acch    1     move the contents of register
                            r4 to accumulator high
move    r5 >> accl    1     move the contents of register
                            r5 to accumulator low
mula  (r0+1)(r1+1)          multiply the contents of the address
                      3     pointed by r0 and r1 and post inc
                            them and add to the contents of the
                            accumulator
muls  (r0-1) r1             multiply and sub
move    acch >> r4    1     move the contents of the accumulator
                            high to register r4 (real part )
move    accl >> r5    1     move the contents of the accumulator
                            low to register r5 (real part )

move    r6  >> acch   1     move the contents of the register
                            r6 to the accumulator high
move    r7  >> accl   1     move the contents of the register
                            r7 to the accumulator low
mula  (r0+1)(r1-1)          multiply and add
                      3
mula  (r0+1)(r1+2)          multiply and add
move    acch >> r6    1     move the contents of the accumulator
                            high to register r6 (imaginary part)
move    accl >> r7    1     move the contents of the accumulator
                            low to register r7 (imaginary part)
```

arithmetic for data structures, such as circular buffers that occur frequently in high-speed signal processing applications. And lastly signal processing algorithms exhibit a substantial amount of parallelism that is not effectively exploited in a uniprocessor system. These signal processing algorithms are extremely computation intensive requiring on the order of N^3 or N^4 multiplications for each set of date. Clearly, orders of magnitude increases in computation rate are required for real time implementation of these advanced algorithms.

Despite the tremendous growth in digital integrated circuits over the last decade, one cannot simply look to further advances in device fabrication to satisfy high computation need. It has been concluded [5] that barring any unseen breakthroughs in signal processor implementation technologies, the orders of magnitude throughput gains necessary for real time computation of signal processing algorithms must come from architectural advances, i.e., the efficient utilization of parallelism in computation.

The most straightforward approach to parallel signal processing architectures is to connect a number of CPUs to a common bus. However, performance improves linearly with the number of processors only up to the point that bus contention problems become the limitations. Minsky's famous conjecture is, that for a broad range of algorithms, the conflict between N processors for access to shared resources along the common bus limits the performance improvement to $\log_2 N$. Modern "supercomputer" designers have utilized a number of parallel-processing

stratagems to improve on this state of affairs and are achieving performance improvements commensurate with Amdahl's law: namely, $N/\log_2 N$.

This traditional design of parallel computer languages suffers from heavy supervisory overhead incurred by synchronization, communication, and scheduling tasks. These severely hamper the throughput rate which is critical to real time signal processing. These problems led to the development of special purpose systems, Systolic and Wavefront, which yield a perfectly efficient performance improvement factor of N. This is shown in figure 2.3

Of all the parallel architectures, Systolic and Wavefront architectures are the most promising. They provide a combination of characteristics for utilizing VLSI/VHSIC technology for real time signal processing. The modular parallelism with throughput directly proportional to the number of cells, simple control, synchronous data flow, and local interconnects, provides the sufficient versatility for implementing the matrix operations needed for signal processing applications.

In the next section Systolic and Wavefront VLSI computing structures are introduced. The timing analysis for the above architectures are also briefly considered. In the final section, the nature of interface required by the Systolic or Wavefront array to be integrated into a larger system is presented.

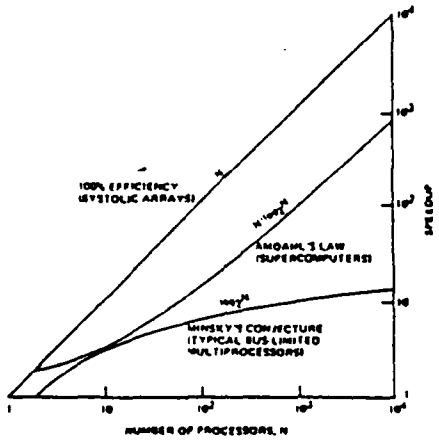


Figure 2.3 Speedup by advanced architecture with N processors working concurrently [5]

2.3.1 Systolic array architecture

VLSI processing structures are suitable for implementing 'compute - bound' algorithms rather than 'input/output - bound' computation. In a compute bound algorithm, the number of computing operations is larger than the total number of input and output elements. Otherwise, the algorithm is input/output bound. Any attempt to speed up an input/bound computation must rely on an increase in memory bandwidth. Memory bandwidth can be increased by the use of either fast components (which could be expensive) or interleaved memories (which could create complicated memory management problems). Also the input/output bound problems are not suitable for VLSI because VLSI packaging must be constrained within a limited number of input/output pins. A VLSI device must balance its computation burden with the input/output bandwidth. *Speeding* up a compute bound computation, however may be accomplished in a relatively simple and inexpensive manner, that is, by the Systolic/Wavefront approach.

This subsection overviews the basic principle of Systolic architectures [6]. A Systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Information in a Systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the "boundary cells".

The basic principle of a Systolic array is illustrated in figure 2.4. By replacing a single processing element with an array of Processing elements (PE's), higher computation

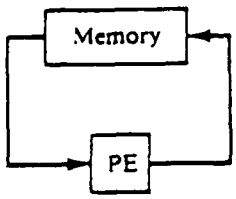


Figure 2.4(a) The conventional processor [8]

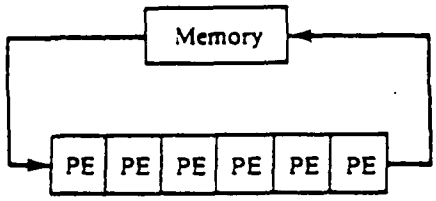


Figure 2.4(b) A systolic processor array [8]

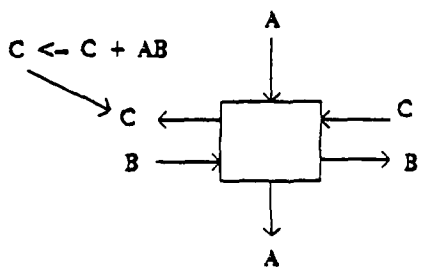


Figure 2.5 "Inner product step processor" [7]

throughput can be achieved without increasing memory bandwidth.

Suppose that input/output bandwidth between a host and a special purpose system is 10 million bytes/sec. Assuming that at least 2 bytes are read from or written to the host for each operation, the maximum rate will be only 5 million operations/sec. (figure 2.4(a)), no matter how fast the special purpose system can operate. The Systolic processor array in figure 2.4(b) will result in 30 MOPS performance with the same input/output bandwidth and assumptions. This approach ensures that once a data item is transferred out of the memory it will be used effectively at each cell.

The basic processing cell used in construction of Systolic arithmetic arrays, is the so-called "inner product step processor" (figure 2.5), the processor performs the operation C <- C + AB (the new value of C is the sum of the old value of C and the product of A and B) and has three registers Ra, Rb, and Rc. Each register has two connections, one for input and one for output. A basic time unit is defined in terms of the operation of this processor. During every time unit interval, the processor shifts in data on its input lines denoted by A,B, and C into registers Ra, Rb, and Rc, respectively. The processor computes Rc<-Rc + RaRb; and makes the input values for Ra and Rb together with the new value of Rc available as output on the output lines denoted by A, B, and C, respectively. All outputs are latched and the logic is clocked so that when one processor is connected to another, the changing output of one during a unit time

interval will not interfere with the input to another during this time interval.

Illustrated below [7] is the construction of a Systolic array for the multiplication of a matrix $A=(a_{ij})$ with a vector $X = (x_1, \ldots, x_n)^T$. The elements in the product $Y = (y_1, \ldots, y_n)^T$ can be computed by the following recurrence.

$$y_i^{(1)} = 0$$

$$y_i^{(k+1)} = y_i^{(k)} + a_{ik} x_k$$

$$y_i = y_i^{(n+1)}$$

Matrix A is a n * n band matrix with bandwidth w = p + q - 1. (figure 2.6, where p = 2 and q = 3). The above recurrence can be evaluated by pipelining the x and y through w linearly connected processors. As illustrated in figure 2.7 , the Systolic array is made up of 4 processors as w = 4.

The general scheme of a pipelining algorithm can be viewed as follows: the y_i , which are initially zero, move to the left, while the x_i , move to the right. The a_{ij} move down. All the moves are synchronized. It turns out that each y is able to accumulate all its terms, namely, $a_{i,i-2} x_{i-2}$, $a_{i,i-1} x_{i-1}$, $a_{i,i} x_i$, $a_{i,i+1} x_{i+1}$, before it leaves the network. Figure 2.8 illustrates the first seven steps of the algorithm.

The algorithm is now explained in detail. Assume that the

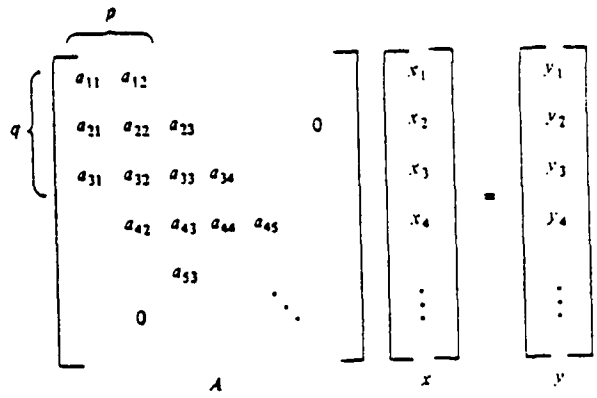


Figure 2.6 Multiplication of a vector by band matrix with p=2 and q=3 [7]

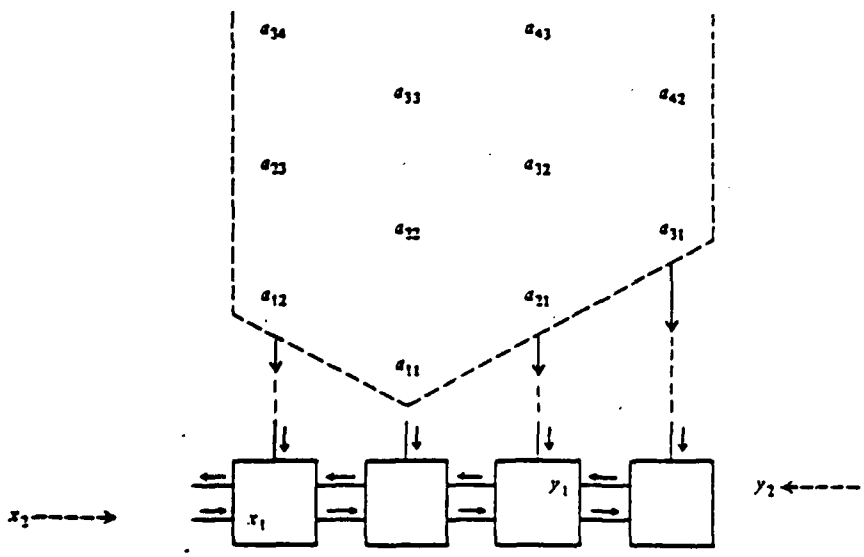


Figure 2.7 Linearly connected network for matrix-vector multiplication [7]

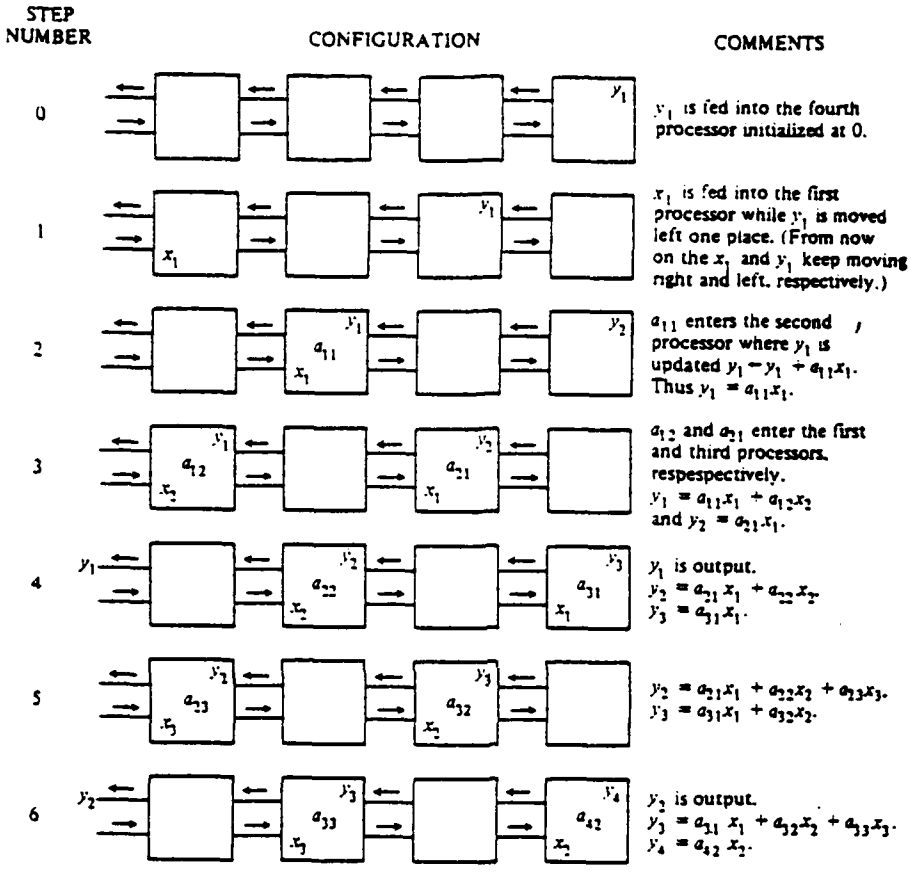


Figure 2.8 First seven steps of the matrix-vector multiplication algorithm [7]

processors are numbered by integers 1,2, .. , w from the left-end processor to the right-end processor. Each processor has three registers Ra, Rx and Ry, which will hold entries in A, x and y, respectively. Initially, all registers contain zeros. Each step of the algorithm consists of the following operations (for odd-numbered time steps, only odd-numbered processors are activated, and for even-numbered time steps, only even-numbered processors are activated):

1.Shift.

Ra gets a new element in the band of matrix A.
Rx gets the contents of register Rx from the left neighboring node. (The Rx in processor 1 gets a new component of x) Ry gets the contents of register Ry from the right neighboring node.

2. Multiply and Add.

Ry <- Ry + Ra Rx

Using the type inner product step processor, we notice in the above example the three shift operations can be done simultaneously. Suppose the bandwidth of A is w. It is readily seen that after w units of time the components of the product y = Ax start shifting out from the left-end processor at the rate of one output every two units of time. Therefore, using this network all the n components of y can be computed in 2n+w time units. This can be compared to the O(wn) time needed for the sequential algorithm on a uniprocessor.

All the moves in the Systolic array are synchronized and

data is clocked into each cell with reference to a global clock. As with all synchronized systems, the Systolic array suffers from the problem of clock skew.

VLSI Systolic arrays can assume many different structures for different compute-bound algorithms. Figure 2.9 [8] shows various Systolic array configurations. Their potential application in performing those computations is listed in Table 2.1 [8].

2.3.2 Wavefront array architecture

Since data movements are controlled by global timing reference beats in the Systolic array, speed variations in computations will cause the array to be clocked at the rate of the slowest computation thereby introducing extra delays.

A simple solution [9],[10] to the above mentioned problem is to have data movements in an asynchronous fashion. This permits a self-timed or asynchronous distributed control approach to the design of VLSI structures. This concept is the basis for Wavefront VLSI computing structures. Most of the algorithms to be implemented in VLSI structures involve repeated application of relatively simple operations with regular localized data flow in a homogenous computing network. The recursive nature of the algorithm, in conjunction with the localized data dependency, points to a continuously advancing wave of data and computational activity. The computational sequence starts with one element and propagates through the processor array, closely resembling a physical wave phenomenon,

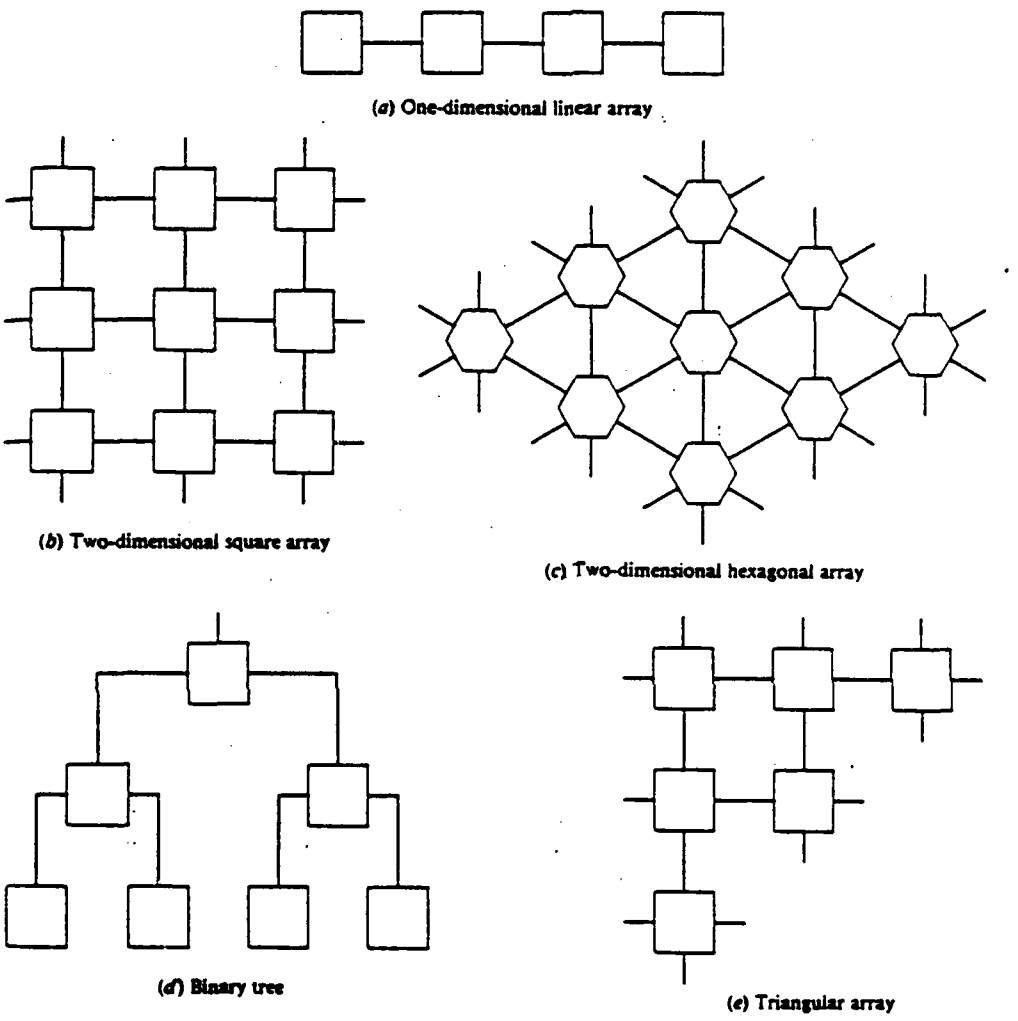


Figure 2.9 Various systolic array configurations [8]

Table 2.1 Computation functions and desired VLSI structures [8]

Processor array structure	Computation functions
1-D linear arrays	Fir-filter, convolution, discrete Fourier transform (DFT), solution of triangular linear systems, carry pipelining, cartesian product, odd-even transportation sort, real-time priority queue, pipeline arithmetic units.
2-D square arrays	Dynamic programming for optimal parenthesization, graph algorithms involving adjacency matrices.
2-D hexagonal arrays	Matrix arithmetic (matrix multiplication, L-U decomposition by Gaussian elimination without pivoting, QR-factorization), transitive closure, pattern match, DFT, relational database operations.
Trees	Searching algorithms (queries on nearest neighbor, rank, etc., systolic search tree), parallel function evaluation, recurrence evaluation.
Triangular arrays	Inversion of triangular matrix, formal language recognition.

(see Figure 2.10). A Wavefront process consists of three steps: (1) the algorithms are expressed in terms of a sequence of recursions; (2) each recursion is mapped to a corresponding Wavefront; and (3) the Wavefronts are successively pipelined through a Wavefront configuration.

All algorithms that have locality (local data-dependent and local control flow) and recursivity (in a recursive algorithm, all processors do nearly identical tasks, and each processor repeats a fixed set of tasks on sequentially available data) will exhibit this wave phenomenon. The main advantages of Wavefront concept are:

1. The Wavefront notion drastically reduces the complexity in the description of parallel algorithms. The mechanism provided for this description is a special-purpose Wavefront-oriented language. Rather than requiring a program for each processor in the array, this language allows the programmer to address an entire front of processors.

2. The Wavefront notion leads to a Wavefront based architecture which preserves Hygens's principle and ensures that the Wavefronts never intersect. Therefore a Wavefront architecture can provide asynchronous waiting capability and consequently can cope with timing uncertainties, such as local clocking, random delay in communications, and fluctuations of computing times.

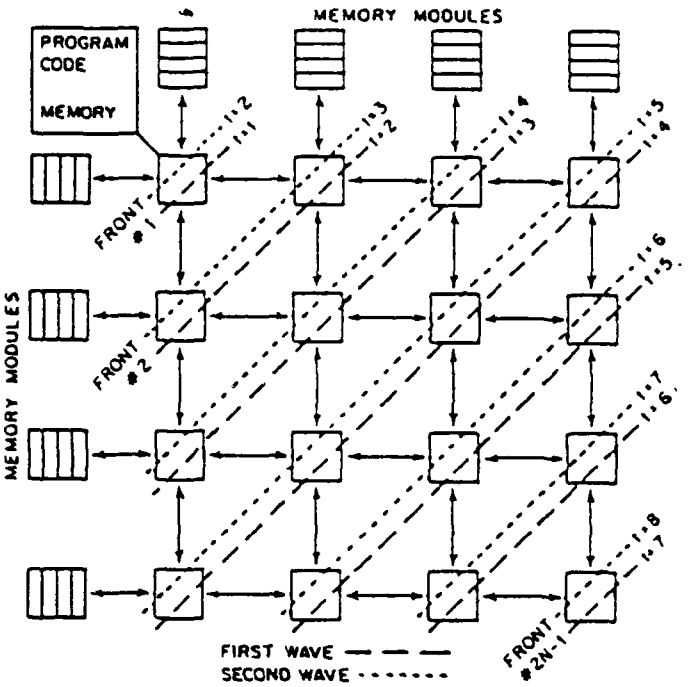


Figure 2.10 WAP configuration [10]

Here an example of the concept of computational Wavefront is given using matrix multiplication. The topology of the matrix multiplication algorithm can be mapped onto the square N * N matrix array of the Wavefront Array Processor (WAP), as in Figure 2.10. Successive pipelining of the Wavefronts through the computational array will accomplish the computation of all recursions.

Let A and B be NxN matrices, define C as the product of A and B, i.e., C = AxB. Clearly, C is a NxN matrix also. The matrix A can be decomposed into columns A and matrix B into rows B , and the matrix C can be formed as,

$$C = A_1 * B_1 + A_2 * B_2 + \ldots\ldots\ldots + A_N * B_N$$

where the product $A_i * B_j$ is termed "outer product". The matrix multiplication can then be carried out in N recursions (each executing one outer product).

$$C^{(k)} = C^{(k-1)} + A_k * B_k$$

There will be N sets of computational "Wavefronts" involved, one for each recursion. More explicitly,

$$c_{ij}^{(k)} = c_{ij}^{(k-1)} + a_i^{(k)} b_j^{(k)}$$

for k = 1,2,....,N. For simplicity, let

$$a_i^{(k)} = a_{ik} \quad , \quad b_j^{(k)} = b_{jk}$$

The computational Wavefront for the first recursion in matrix multiplication is now examined further. Suppose that the registers of all the processing elements (PE's) are initially set to zero:

$$c_{ij}^{(k)} = 0 \quad , \quad \text{for all } (i,j)$$

The entries of A are stored in the memory modules to the left (in columns), and those of B in the memory modules on the top (in rows). The process starts with PE(1,1) and

$$c_{11}^{(1)} = c_{11}^{(0)} + a_{11} \; b_{11}$$

is computed. The computational activity then propagates to the neighboring PE's (1,2) and (2,1), which will execute:

$$c_{12}^{(1)} = c_{12}^{(0)} + a_{11} \; b_{12}$$

and

$$c_{21}^{(1)} = c_{21}^{(0)} + a_{21} \; b_{11}$$

The next front of activity will be at PE's (3,1),(2,2) and (1,3), thus creating a computation Wavefront traveling down the processor array. Once the Wavefront sweeps through all the cells, the first recursion is complete. As the first wave propagates, an identical second recursion can be executed in

parallel by pipelining a second Wavefront immediately after the first one. For example the (1,1) processor will execute

$$c^{(2)} = c_{11}^{(1)} + a_{12} \quad b_{21}$$

.

.

.

$$c_{ij}^{(K)} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + \ldots + a_{iK} * b_{Kj}$$

and so on.

The pipelining of the Wavefronts are implemented in a highly asynchronous ashion [11]. The processors in the array must wait for a primary Wavefront (of data), then perform the computation it calls for and, finally, act as a secondary source of new Wavefronts. For example operations ($c_{12}^{(1)}$) and ($c_{21}^{(1)}$) will not be executed until PE's (1,2) and (2,1) confirm receipt of (a_{11} , b_{12}) and (a_{21} , b_{11}) respectively. For the same reason, in the next front of the wave, cells (1,3), (2,2) and (1,3) will be involved. PE (2,2) for example, has to wait until PE's (1,2) and (2,1) flow their data, b_{12} and a_{21} , respectively. Only after the arrival of that data will the (2,2) cell execute its own operation ($c_{22}^{(1)}$) = $c_{22}^{(0)} + a_{21}\ b_{12}$ and activate its own successors, PEs (2,3) and (3,2).

One of the main advantages of the Wavefront array processor

is the flexibility in programming.

2.3.2.1 Matrix Data Flow language

This section explains a Wavefront-oriented language for programming the WAP. This Wavefront language is tailored toward the description of computational Wavefronts and corresponding data flow for a large class of algorithms. Since matrix algorithms are typical of this class the language is named Matrix Data Flow Language (MDFL)[10].

There are two approaches to programming the WAP: (1) a local approach describing the actions of each processing element, and (2) a global approach describing the actions of each Wavefront. To allow the user to program the WAP in both these fashions, two versions of MDFL are present: global and local MDFL.

Here a brief introduction to the programming methodology of Wavefront array processors is given using MDFL. The most straightforward method of programming the WAP is to specify the actions of each Wavefront at each of its (2n -1) positions (fronts), (see Figure 2.10). Nevertheless, because of the regularity and recursivity in almost all matrix algorithms one can assume the following.

1) Space Invariance:

   In a particular kind of processor the tasks performed by a Wavefront must be identical at all (2n-1) Wavefronts.

2) Time Invariance:

   Recursions are identical.

Accordingly, global MDFL provides two repetitive constructs, the space repetitive constructs,

WHILE WAVEFRONT IN ARRAY DO

BEGIN <TASK T> END

so that task T is repeated at all fronts, and the time repetitive construct

REPEAT <ONE RECURSION> UNTIL TERMINATED

so that the same recursion is repeated.

The REPEAT construct is inherently concurrent in that successive Wavefronts are pipelined through the array. As soon as the kth Wavefront is propagated, the (1,1) processor initiates the (k+1)st Wavefront.

To allow for more than one Wavefront per recursion, the complete global MDFL program will have the syntax

```
BEGIN
 SET COUNT < >;
 REPEAT
    < TASKS A >;
    WHILE WAVEFRONT IN ARRAY DO
      BEGIN
        < TASKS B>;
      END;
    WHILE WAVEFRONT IN ARRAY DO
      BEGIN
        < TASKS C >;
      END;
    < TASKS D >;
    DECREMENT COUNT;
 UNTIL TERMINATED;
ENDPROGRAM.
```

Each recursion will execute the instructions within the REPEAT .. UNTIL construct. The number of recursions is set by SET COUNT. In this example, a recursion consists of two Wavefronts. At the start, tasks A are performed only at the (1,1)

processor. The first Wavefront of each recursion will perform tasks B at each of its (2n-1) fronts. The second wave will execute tasks C in each of these fronts immediately after tasks B have been concluded. When the count becomes zero, TERMINATED is set and a 'phase' of identical recursions is over.

The corresponding local MDFL program for interior processors will be

```
REPEAT
   < TASKS B >
   < TASKS C >
UNTIL TERMINATED;
```

where B and C are the compiled versions of B and C. The conversion of global program to local version is fairly easy.

MDFL notion makes it possible to program an array of asynchronous processors in a simplistic fashion using the MDFL language which are modular and easy to follow.

2.3.3 Timing analysis of Systolic and Wavefront array architecture

The timing framework [11] is a very critical issue in designing the system especially when one considers large scale computational tasks. Two opposite timing schemes come to mind, namely the synchronous and the asynchronous timing approaches. The Systolic array is an example of a totally synchronous system, and Wavefront array is an example of an asynchronous timing system.

2.3.3.1 Timing analysis of Systolic arrays

As the Systolic array is wholly synchronous, it requires global clock distribution. Therefore, different processing elements receive clock signals by different paths. The elements may not receive clocking events at the same time. Synchronization failure can result from these clock skews. These synchronization failures can be avoided by lowering clock rates and by adding delay to circuits, thereby slowing the computation. Unless operating at possibly unacceptable reduced speeds, very large systems controlled by global clocks can be difficult to implement because of the inevitable problem of clock skews and delays.

Also, because of the strict synchronized timing, all of the PEs except some special peripheral elements must be performing the same task in unison. There is no room for multitasking, even if the tasks are serial and not interwoven one into the other.

The "clock skew" phenomenon arises due to three factors:

1) The resistance/capacitance (RC) of the global clock distribution line;

The line capacitance plays a dominant role in the line delay. In fact, when the signal of interest is the clocking signal, it will, in many cases create disastrous synchronization problems, particularly if the clock traversed a large distance on the chip. This could commonly occur in a large VLSI chip. The resistance too, is dependent on the length of the line. Due to constraints involved in the layout of the

VLSI chip, a major factor in line resistance is the distribution of the material of which the line is made. As the resistance of diffusion is on the order of 100 times that of metal, the length of diffusion paths in the clock line is the predominant factor in that line's time constant.

2) The unequal clock paths to various PEs in the array;

The clock skew due to unequal clock path lengths to the PEs may become potentially hazardous. To eliminate this skew contributor for a square array for example, a H-tree clocking network is often used. A H-tree is implemented by placing the global clock generator at the root of a binary distribution tree. All the processing elements are at the various levels of the tree as children of preceding nodes. Every node represents a processor; this ensures that all clock paths are of equal length to each processor. This clock distribution scheme appears to be optimal for square arrays.

3) The variance of values of the gate threshold voltage;

The PE gate receives and generates the global clock signal to the interior of the PE, thus serving as a buffer between the global clock distribution network and the local clock distribution paths. The uncertainty of the threshold voltage of this gate which arises due to fabrication phenomenon contributes to the clock skew.

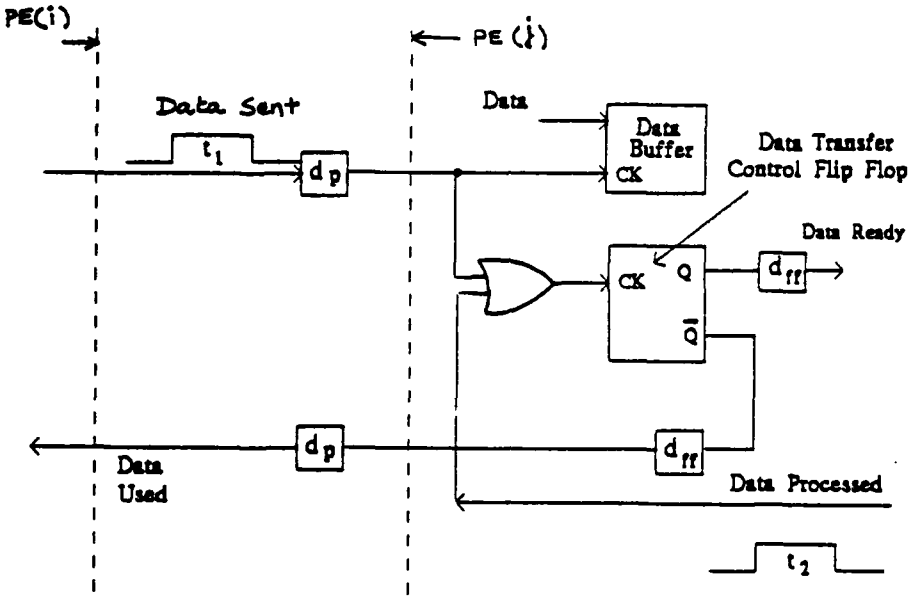
2.3.3.2 Timing analysis of Wavefront arrays

The self-timed asynchronous [11] scheme of the Wavefront array can be costly in terms of extra hardware and delay in each cell. It has the advantage however, that the time required for a communication event between two cells is independent of the size of the entire processor array. An advantage that self-timed systems often enjoy, in addition to the absence of clock-skew problems, is a performance advantage that results from each cell being able to start computing as soon as its inputs are ready and to make its outputs available as soon as it is finished computing. This allows the array to take advantage of variations in component speed or data dependent conditions allowing faster computation. As data transfer is asynchronous, multi-tasking, where cells in array perform different kinds of operation is possible in Wavefront arrays.

The Wavefront arrays employ a handshaking protocol to transfer data between the adjacent PEs. The protocol ensures the regularity and continuity of the flow of information through the processor array. This calls for additional number of input and output signal lines for each PE element, thereby increasing complexity and hardware cost.

The global clock of the synchronous scheme is now replaced by the data-sent and data-used lines which establish an exchange of information between the adjacent processors with regards to data transmission timing. This handshaking operation is shown in Figure 2.11. The transfer of data in the basic model from PE(i) to PE(j) calls for PE(i) to apply the

Figure 2.11 Interprocessor handshaking scheme [11]



appropriate data to the interprocessor data bus and a pulse generated on the data-sent control line. The width of that pulse, t(1) must be greater than the data setup time of the data input buffer of PE(j). After the pulse has been generated, PE(i) can turn to the next task of the recursion. The negative edge of the data-sent pulse enters the data into the data input buffer of PE(j) and also toggles the data transfer control flip flop, thereby notifying PE(j) of the availability of data. In general, PE(j) will be waiting for that data and will immediately execute a Fetch instruction. In the worst case, the time lost in this transaction is, T(pe(j)) + d(ff), where d(ff) is the input to output delay involved in toggling the data control transfer flip flop (see Figure 2.11). Upon completion of the data fetch, PE(j) issues a data processed pulse of duration t(2). t(2) must be larger than the clock pulse width required by the flip flop transition. This involves a time delay of d(ff) + d(p), where d(p) is the propagation delay of the interprocessor flagging signal.

It is to be noted that not much time is lost in handshaking process, as after a FLOW has been executed, PE(i) is implementing its next task concurrently with the FETCH executed by PE(j). By the time PE(i) has to carry out the next FLOW to PE(j), the Data-Used signal will have already been set, and there is no waiting involved. The timing penalty for this situation is, at worst, t(2) + d(ff). This penalty is paid only once, and does not multiply by the number of recursions, nor by the number of FLOW(FETCH) tasks, provided they are not

consecutive.

2.3.4 Implementation considerations of Array processor systems

Figure 2.12 [12] depicts a possible overall array processor system configuration. The design considerations for the major components are described in detail below.

2.3.4.1 Host computer

The host computer should: provide batch data storage, management and formatting, determine and schedule program that controls the interface system and connection network, and generate and load object codes to the PE's. The host selected should be capable of interfacing with the high speed array processor units which have a high input/output bandwidth.

2.3.4.2 Interface system

The interface system consists of buffer memory and the control unit. The interface system, connected to the host, via the host bus, has the functions of downloading data and up loading data. Based on the schedule program, the control unit monitors the interface system and array processor. The interface system should also furnish an adequate hardware support for many common data management operations. In addition to handling bus protocols, the interface system generates addresses for buffer memory accesses, controls the loading of the buffer memory, and schedules and monitors the computations carried out by the PE array.

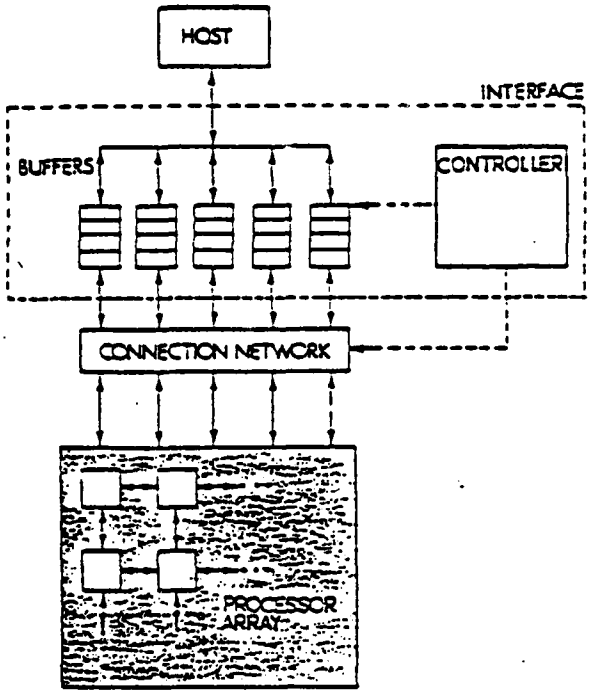


Figure 2.12 Array processor system configuration [9]

These memory units are used as buffers between the low bandwidth host bus and the special high bandwidth buses in the interface system. By holding data that are to be used repeatedly by the Systolic arrays in this memory, the arrays can proceed with high speed, without consuming much host-bus bandwidth.

2.3.4.3 Connection network

Connection networks provide a set of mappings between processors and memory modules which accommodate certain common global communication needs. Incorporating certain structured interconnections may significantly enhance the speed performance of the processor arrays.

2.3.4.4 Processor arrays

Processor arrays consist of the PEs in a particular arrangement to solve the specific algorithm. When the problem is decomposed into many subproblems, each of these subproblems can be run in parallel in different processor arrays. In this manner the connection network can be used to facilitate the data pipelining between the arrays, and thereby increasing the overall processing speed by one more order of magnitude.

2.4 Chapter summary

In this chapter we presented three different hardware architectures 1) the general purpose microprocessor architecture, 2) the digital signal processor (DSP) architecture and 3)

the VLSI architecture.

The microprocessor architecture does not have its arithmetic logic unit optimized to the multiply-add computation which is often encountered in signal processing algorithms. The microprocessor chosen for analysis was a Mc 68020 which has a cycle time of 60ns and performs a complex multiply-add operation at the rate of 0.5 MOPS. The digital signal processor architecture has its arithmetic logic unit (ALU) optimized to compute multiply-add operation. The DSP chosen for analysis was a LM 32900 which has a cycle time of 100ns and performs a complex multiply-add operation at the rate of 5.7 MOPS. An important difference between these two architectures is the ratio of the multiply to add time. In the microprocessor chosen, the multiply to add time ratio is on an average 6:1, but in the digital signal processor, a multiply takes the same time as an add. Thus, if the algorithm considered has many isolated additions to be performed, then the microprocessor architecture will be more suited than the DSP architecture.

Next the characteristics of the VLSI architectures, specifically Systolic and Wavefront architectures were discussed. It was observed that VLSI architectures are suited for compute bound operations rather than input-output bound operations. Also presented was a timing analysis for the two VLSI architectures. Systolic architectures operate in a synchronized manner and thus suffer from clock skew problems. The Wavefront architecture is free from clock skew problems as it is self-timed but requires extra hardware.

For implementation of HF adaptive antenna signal processing algorithms the following chapters evaluate the architectures considered in this chapter.

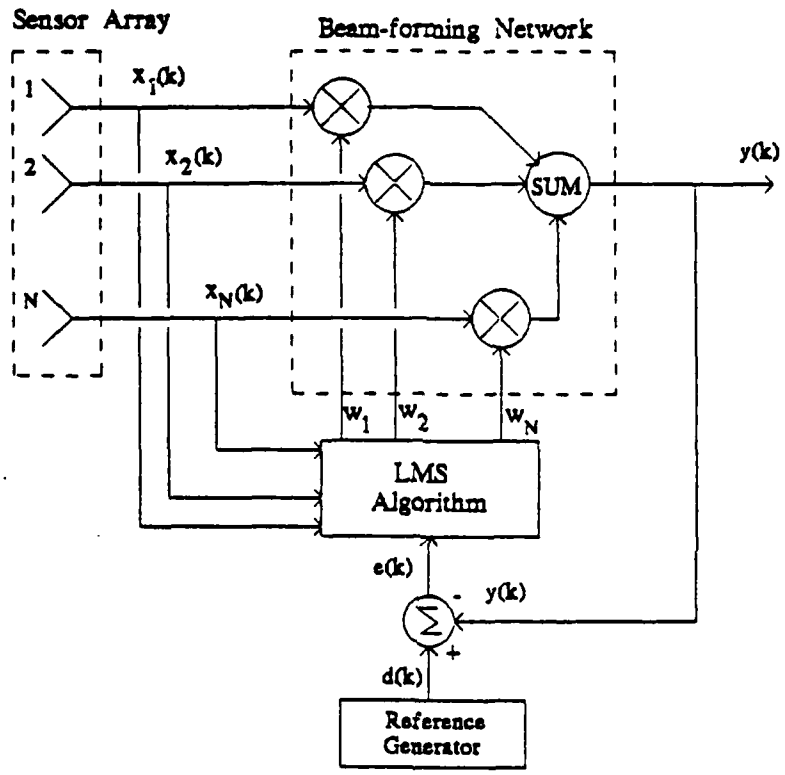
3.0 HARDWARE ARCHITECTURES FOR LMS ALGORITHM

The purpose of this chapter is to discuss the various hardware implementations of the LMS algorithm, to study the feasibility of these architectures, and to recommend suitable configurations. We present a brief introduction to the LMS algorithm in section 3.1 and indicate the equations that are to be solved by the hardware implementation. An initial loading analysis procedure is provided for the LMS algorithm. Section 3.2 provides a method for the analysis to determine an initial estimate on the computational loading required by different functions constituting the algorithm. The loading analysis is then performed for various hardware architectures. The hardware architectures considered are the general purpose microprocessor, the digital signal processor and the VLSI architecture and are discussed in sections 3.3, 3.4 and 3.5 respectively. A summary discussing the feasibility of these architectures and a recommended architecture is discussed in section 3.6.

3.1 LMS algorithm

The discussion of the LMS algorithm [13] begins with an explanation of the Mean Square Error (MSE) performance criterion. The LMS controlled adaptive array system is shown in Figure 3.1 and will be used to present the concept behind the MSE performance criterion. In the adaptation process the weight vector of the linear combiner is adjusted so as to cause the output y(k) to agree as closely as possible with the desired response signal, d(k). For the present, the desired signal is

Figure 3.1 LMS-controlled Adaptive Array System



available. An error signal is expressed as

$$e(k) = d(k) - y(k) \qquad (3.1)$$

where y(k) is the linear combination of the input samples $\underline{X}(k)$ and the weights W(k).

$$e(k) = d(k) - \underline{W}^{T}\,\underline{X}(k) \qquad (3.2)$$

The subscript 'k' from the weight vector is removed because, in this discussion, the weights are not adjusted. The instantaneous squared error is obtained by squaring equation (3.2)

$$e^{2}(k) = d^{2}(k) - 2\,d(k)\,\underline{X}^{T}(k)\,\underline{W} + \underline{W}^{T}\,\underline{X}(k)\,\underline{X}^{T}(k)\,\underline{W} \qquad (3.3)$$

Assuming statistical stationarity for e(k), d(k) and X(k), the expected value of equation (3.3) provides the MSE as

$$E[e^{2}(k)] = E[d^{2}(k)] + \underline{W}^{T}\,E[\underline{X}(k)\underline{X}^{T}(k)]\,\underline{W} - 2\,E[d(k)\,\underline{X}^{T}(k)]\,\underline{W} \qquad (3.4)$$

This equation represents the mean square error as a function of the weights.

Equation 3.4 can be expressed in a more convenient form as follows. Let matrix R be defined as the 'input correlation matrix.

$$R = E[\underline{X}(k)\,\underline{X}^{T}(k)] \qquad (3.5)$$

The main diagonal terms of R are the mean square of the input components. The cross terms are the cross correlation among the input components. Let column vector P be defined as the cross correlation between the desired response and the sensor element output, i.e.,

$$P = E \begin{bmatrix} d(k)\, X_1(k) \\ \vdots \\ d(k)\, X_N(k) \end{bmatrix} \qquad (3.6)$$

Letting the mean square error be designated as 'ξ' equation (3.4) can be expressed as

$$MSE = \xi = W R W - 2 P W + E[d(k)] \qquad (3.7)$$

The mean square error 'ξ' is precisely a positive quadratic function of the weights. The vertical axis represents the mean square error and the horizontal axis the values of the weights. The error function results in a bowl shaped surface. This paraboloid is called the 'performance surface'. It contains no local minima and for two weights is shown in Figure 3.2.

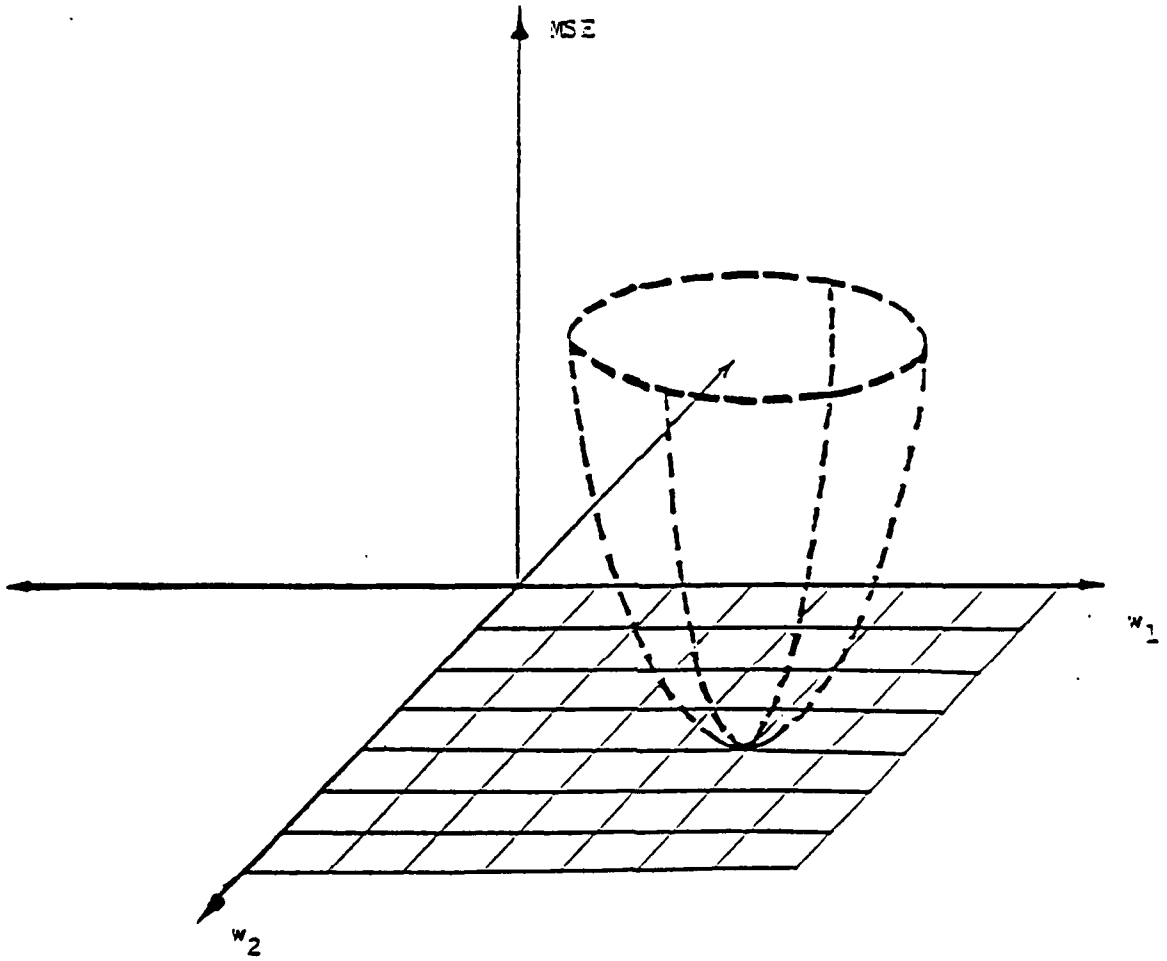
The point at the bottom of the bowl, when projected onto the weight-vector plane, gives the optimal weight vector, W^*. The LMS algorithm attempts to estimate this optimum weight vector.

The LMS algorithm is an implementation of the method of the steepest descent. Using this method, the updated weight vector is equal to the past weight vector plus a change that is proportional to the negative gradient. This is expressed as:

$$W(k+1) = W(k) - \mu \nabla \qquad (3.8)$$

where μ is the gain constant which regulates the stability and speed of adaptation and ∇ is the estimated gradient of the mean square error performance surface. Assume that μ has been chosen such that proper performance specifications are met. Updating the weights can be thought of as descending

Figure 3.2 MSE Performance Surface



along the aforementioned performance surface in an attempt to reach the 'bottom of the bowl'. By using the square of a single error sample instead of the MSE the LMS algorithm estimates the gradient as,

$$\hat{\nabla} = -2\, e(k)\, \underline{X}^{*}(k) \qquad (3.9)$$

Replacing for ∇ in equation (3.8), yields the LMS algorithm equation as

$$W(k+1) = W(k) + 2\mu\, e(k)\, \underline{X}^{*}(k) \qquad (3.10)$$

As the weight changes in each iteration are based on imperfect gradient estimates, the adaptive process does not follow the true line of steepest descent on the performance surface. Realization of the above algorithm using various hardware architectures is the main purpose of this chapter.

3.2 Loading analysis

The loading analysis involves determining four basic system parameters [14],

1. Input requirements
2. Function execution time budgets
3. Computational loading estimate
4. Memory requirements

3.2.1 Input requirements

The LMS algorithm implementation involves solving equation

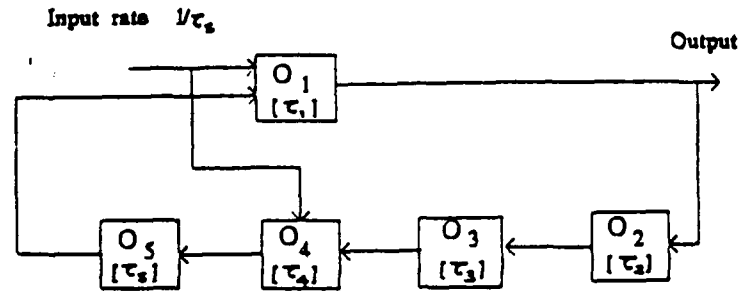
(3.10) to update the weights. The equation is broken down into manageable functions as indicated in Figure 3.3.

As can be seen from the Figure 3.3, the algorithm functions have been partitioned into a forward loop and a feedback loop. The computations required in the feedback loop must be completed before the next input sample is applied to the loop. In other words, the processing performed for the current input must be completed before the next input sample is processed. In general if the adaptive algorithm constituted of computations [15], O_1 through O_M in the forward loop and O_{M+1} to O_{M+N} in the feedback loop, then the structure of the adaptive algorithm would be as shown in Figure 3.4. As each computation introduces a latency of τ_j for the j-th computation, it is necessary for the sum of these latencies to be less than the sample period τ_s. This sampling period, τ_s, is dictated by constraints imposed by the HF channel characteristics and the nature of the system in which the LMS algorithm is incorporated. These constraints establish a minimum computation time for the hardware. The sample period τ_s for the LMS algorithm is obtained as follows.

3.2.1.1 HF channel requirements

The adaptation process consists of the computation and update of the weights until the optimal weights are obtained. This gives the minimum mean square error. The objective has to been to move on the bowl shaped performance surface towards the 'bottom of the bowl'. There is an important property of this

Figure 3.3 Partitioning of LMS algorithm for hardware realization



O_1 : $W^T X(k) = Y$

O_2 : $Y - d(k) = e$

O_3 : $\mu' e = e'$ where $\mu' = 2\mu$

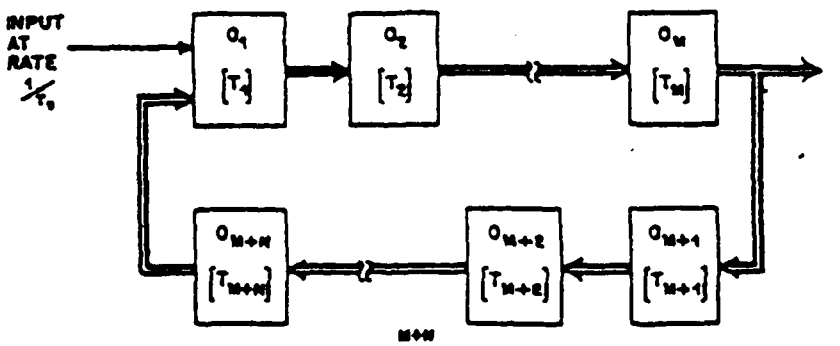
O_4 : $e' X^*(k) = B$

O_5 : $W(k) + B = W(k+1)$

Latency constraint

$$\sum_{i=1}^{5} \tau_i \leq \tau_s = 1073\,\mu s$$

Figure 3.4 Feedback loop and lower limit on arithmetic speed [15]



LATENCY CONSTRAINT $\sum_{j=1}^{M+N} T_j \leq \tau_s$

T_j = EXECUTION TIME FOR OPERATION O_j

performance surface in adaptive signal processing that is to be noted. If the incoming signals are stationary and have invariant statistical properties, then the performance surface remains fixed and rigid in the cordinate system. The adaptation process starts at some point on the performance surface and moves towards the 'bottom of the bowl' i.e., the neighborhood of the minimum mean square error and stays there. Once these optimal weights are determined they need not be changed as the performance surface is fixed.

If the signals are not stationary, then the situation changes [16]. If the statistical properties of the signals change slowly, then the performance surface is slowly moving in its co-ordinate system. Now the adaptation process consists of not only moving downhill along the performance surface towards the minimum, but also tracking the minimum as it moves about in the co-ordinate system.

We are concerned with the signals whose statistical properties slowly vary. In fact, it has been determined that the HF channel can be considered stationary for times in the order of 100 ms. This implies that the incoming signals can be considered stationary i.e., they have invariant statistical properties for 100ms. During the next 100ms the signals are again considered stationary, but the statistical properties have changed with respect to the previous 100 ms. This means that the performance surface has shifted, and thus the adaptation process is to converge to a new set of optimum weights as the minimum has changed.

The process of obtaining a set of optimum weights is considered as one convergence. The number of iterations of the LMS algorithm needed for one convergence is a function of the channel considerations and has been determined by simulation studies [1], (see Table 3.1).

3.2.1.2 System considerations

The adaptive antenna array studied here is incorporated in a direct sequence spread spectrum (DSSS) system. The LMS algorithm can be easily implemented in a DSSS system as discussed by Compton [17]. The problem here is the derivation of the reference signal d(k). The reference signal used in the LMS algorithm must satisfy the following criteria

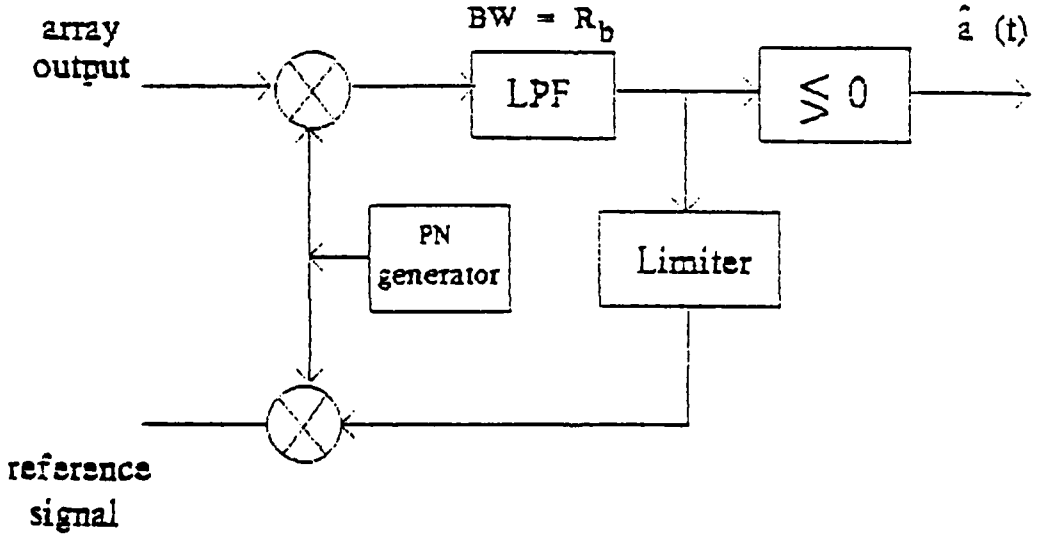
1. The reference signal must be highly correlated with the desired signal at the array output.
2. The reference signal must be uncorrelated with any interference signal components appearing at the array output.

The reference signal generation loop is shown in Figure 3.5. A locally generated PN sequence is mixed with the array output signal. It is assumed that this is the same PN sequence used to spread the information signal, and that its phase is coherent with the phase of the received PN sequence. The mixing operation strips the PN code from the desired information signal. Next the signal is low-pass filtered over the data bandwidth. The limiter then removes any amplitude modulation present. Now the local PN sequence is applied to the limiter

Table 3.1 Convergence properties of LMS algorithm [1]

	Number of iterations required for one convergence	
	Mean	σ
LMS Algorithm	547	171

Figure 3.5 Reference loop



output to respread the data.

For this discussion the system design parameters are as follows.

Modulation scheme	BPSK
Data rate	300 bits/sec
Processing gain	31

The processing gain of 31 implies that the chip rate is 9300 chips/sec. The reference loop unit integrates the received array output Y(k), over a period duration of one bit (= 3.3 ms). As there are 31 chips/bit, this reference loop unit is provided with 31 values of Y(k) every 3.3 ms. This leads to 31 iterations of the LMS algorithm per 3.3ms as every iteration produces a new value of Y(k). Thus 9300 iterations of the LMS algorithm are to be carried out in 1sec or 930 iterations in 100ms.

As previously shown the HF channel characteristic imposes the need for one convergence to be obtained in 100ms. Table 3.1 shows that the LMS algorithm needs 550 +/- 175 iterations per convergence which gives a maximum of 725 iterations per convergence. A design of 930 iterations of the LMS algorithm in 100 ms meets both the HF channel constraint of 725 iterations/100 ms and the system considerations constraint of 930 iterations/100 ms. An upper bound is now set on the sample period (or time per iteration) for the LMS algorithm

$$= 100\text{ms}/930 = 107.5\,\mu\text{s}$$

This implies that the hardware implementing the LMS algorithm must perform one iteration in 107.5 μ s.

3.2.2 Execution time budgets

Once the input requirements are determined, we know that the functions constituting the algorithm have to be calculated once every 107.5 μ s. Thus a portion of this iteration time is to be allocated to each function of the algorithm. To allocate the time budget to each function, we have to determine the number of real operations required by each function. Note that an add operation is equivalent to a subtract operation in this discussion, so all subtract operations are referred as add operations. One complex addition requires 2 real adds, so in total requires 2 real operations. One complex multiplication requires 4 real multiplies and 2 real adds, giving a total of 6 real operations. The following summarizes the requirement for each function (see figure 3.3).

1) $Y = W^{T} X(k)$ (O_1)

   This function constitutes 36 complex multiplications and 35 complex additions and thus requires (36*6) + (35*2 = 286 real operations.

2) $e = Y - d(k)$ (O_2)

   This function involves a complex subtraction and thus requires 2 real operations.

3) $e' = e\mu'$ (O_3)

   This function requires 2 real multiplications as μ' is a real quantity.

4) $B = e' X^{*}(k)$ (O_4)

   This function consists of 36 complex multiplications.

To obtain X^* involves 36 real subtractions. Thus this function requires (36*6) + 36 = 252 real operations.

5) $W = W(k) + B \quad (O_5)$

This function consists of 36 complex subtractions and thus requires (36*2) = 72 real adds.

These observations are tabulated in Table 3.2. The Table also indicates that the LMS algorithm requires 2N complex multiplications and 2N +1 complex additions. Once the number of computations required by each function is determined, then execution time can be budgeted to each function. This procedure is illustrated later for the various hardware architecture considered.

3.2.3 Computational loading

Once the time budget is determined, the computational loading offered by each function is easily obtained by dividing the number of real operations required by the function by the time budget allocated to that function.

3.2.4 Memory requirements

An initial estimate on the memory requirement of the LMS algorithm can be determined as follows. The algorithm requires 2N+1 complex words of storage. X and W each require N words of storage and d(k) requires one word of storage. Because storage of intermediate results is necessary, the memory requirement

Table 3.2 Computational complexity of LMS algorithm

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Complex Multiply	(6) Complex Add
Y	286	144	142	N	N
e	2	-	2	-	1
e'	2	2	-	-	-
B	216	144	108	N	-
W	72	-	72	-	N

for data depends on the hardware architecture considered.

The general structure for the LMS algorithm is as shown in Figure 3.6. The input signals from the N antenna elements are directed to N parallel sections. Note the update of weight of one antenna element is independent of the update of weight of another antenna element. Each of these sections initially performs the multiplication of the weight with the input signal. Then later perform the weight update when the $e'(k)$ is obtained as shown in Figure 3.6. Each section can be allocated to a separate 'processor' or depending on the computational capacity of the processor a few of the sections can be coupled and placed on a single processor (see Figure 3.7).

3.3 Microprocessor Architecture implementation

The general purpose microprocessor architecture will now be evaluated for the LMS algorithm. The microprocessor chosen is Mc 68020 [2], state of the art 32-bit microprocessor, which has a cycle time of 60ns.

The execution time budgets of each function of the algorithm, and computational loading offered by each function is now determined. This also determines the complexity of the algorithm with respect to the microprocessor architecture. The complexity is given as the number of chips needed to implement the algorithm.

Figure 3.6 General structure of LMS algorithm

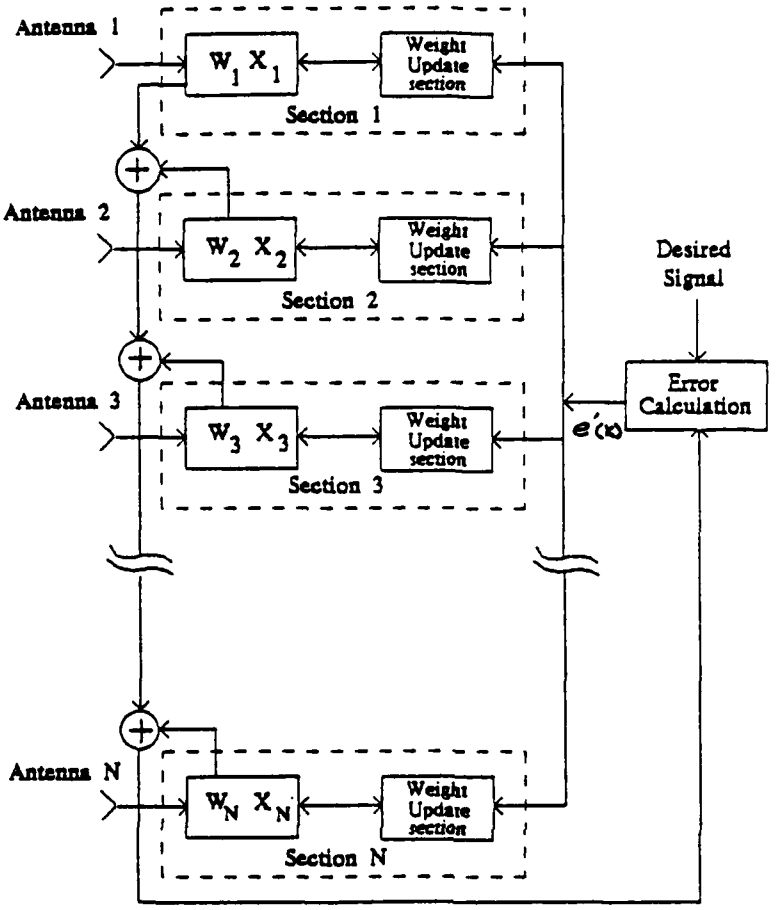
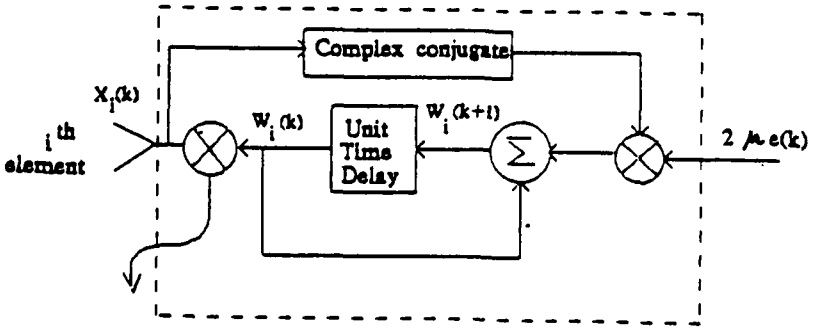


Figure 3.7 Operations performed in one of the sections of figure 3.6



3.3.1 Initial assessment on the Complexity

The complexity of the hardware implementing the algorithm is a function of

1) the number of operations the algorithm needs for an iteration - computational bounds
2) the time allocated for the iteration - input requirements
3) the hardware architecture implementing the algorithm - architecture considerations

When these issues are taken into consideration, then one can obtain an estimate on the number of chips (microprocessors) needed by the system implementing the algorithm. The complexity of the system implementing the algorithm is indicated here as the number of chips needed to construct the system. Loading analysis of the algorithm in the microprocessor architecture environment is now performed.

3.3.1.1 Execution time budgets

The time budget allocated to each function is determined next. Certain factors play a role in determining the complexity of the system for the microprocessor architecture. These factors are:

1. the ratio of time needed to do a multiply operation and the time needed to do an add operation.
2. the speed of the microprocessor with regard to LMS algorithm (millions of operations per second, MOPS).

The summary of the discussion which follows is shown in

Table 3.3. The first column of Table 3.3 contains partitioning of the algorithm. It can be noticed that few functions are coupled as a single task. Functions O_2 and O_3 are coupled together for convenience to form task T_2. The reason for combining functions $B = e^{*} X$ and $W = W(k)+B$ to form task T_3 is that these functions need not be done in sucession but can be interleaved. Once $B = e^{*} X(k)$ is done for antenna element 'i', then the weight can be updated at once by performing $W = W(k) + B$. Processor cycles can be saved as the intermediate result is still in the register.

The second column of Table 3.3 indicates the number of real operations each of the tasks needs. Column 3 and column 4 of Table 3.3 indicate the number of real multiplies and number of real adds each of the tasks requires. The Mc 68020 microprocessor chosen has a multiplication to addition ratio of 6 : 1. Thus the effective number of operations column is obtained by (number of multiplications) * 6 + (number of additions) * 1. The effective number of operations for each task represents the task in terms of adds. These then provide the true complexity of the task when implemented in the microprocessor.

A portion of the sampling time 107.5μs is allocated to perform each task T_1 through T_3 and is given under the 'execution time budget' column. The time distributed to each task is proportional to the effective number of operations of that particular task. For example, for task T_1 the time budget is obtained as

Table 3.3 Assessment on complexity of LMS algorithm using microprocessor architecture

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Effective number of operations	(6) Time Budget (μs)	(7) Computational loading (MOPS)	(8) Number of chips
Task T_1	286	144	142	1006	52.39	5.46	11
Task T_2	4	2	2	14	0.73	5.48	11
Task T_3	324	144	180	1044	54.38	5.95	12

$$\frac{(1006)}{(2064)} * 107.5\ \mu s = 52.39\ \mu s$$

3.3.1.2 Computational loading

Once the execution time budget for each task is obtained then one can determine the computational loading each task demands. Loading demands are indicated in the computational loading column of Table 3.3. The entries in this column are obtained by dividing the unweighted number of operations (entries in column 2 of Table 3.3) by the execution time budget. This gives the number of operations/ sec needed to perform that particular task. For example for task T_1, the estimated computational loading is obtained as

$$286/52.39\ \mu s = 5.46 \text{ MOPS.}$$

Once the computational loading offered by a task is determined, then the number of chips needed to perform this task is determined. The number of chips needed for each task is determined by obtaining the speed of one chip with regard to the LMS algorithm. It can be observed from Table 3.3 that tasks T_1 and T_3 put together consume the major portion of the iteration time. The operations needed to perform task T_1 and task T_3 thus constitute the 'operation mix' for the LMS algorithm. A complex multiply and add is the basic operation that is being repeated to perform task T_1 and T_3. The 'operation mix' for the LMS algorithm constitutes a complex

multiply and add which involves 8 operations (4 multiplies and 4 adds). The Mc 68020 microprocessor performs these 8 operations at the rate of 0.5 MOPS.

Using this information the number of microprocessors needed for a task can be determined. The number of microprocessors for task T_1 is obtained as

$$5.46 / 0.5 = 11 \text{ microprocessors.}$$

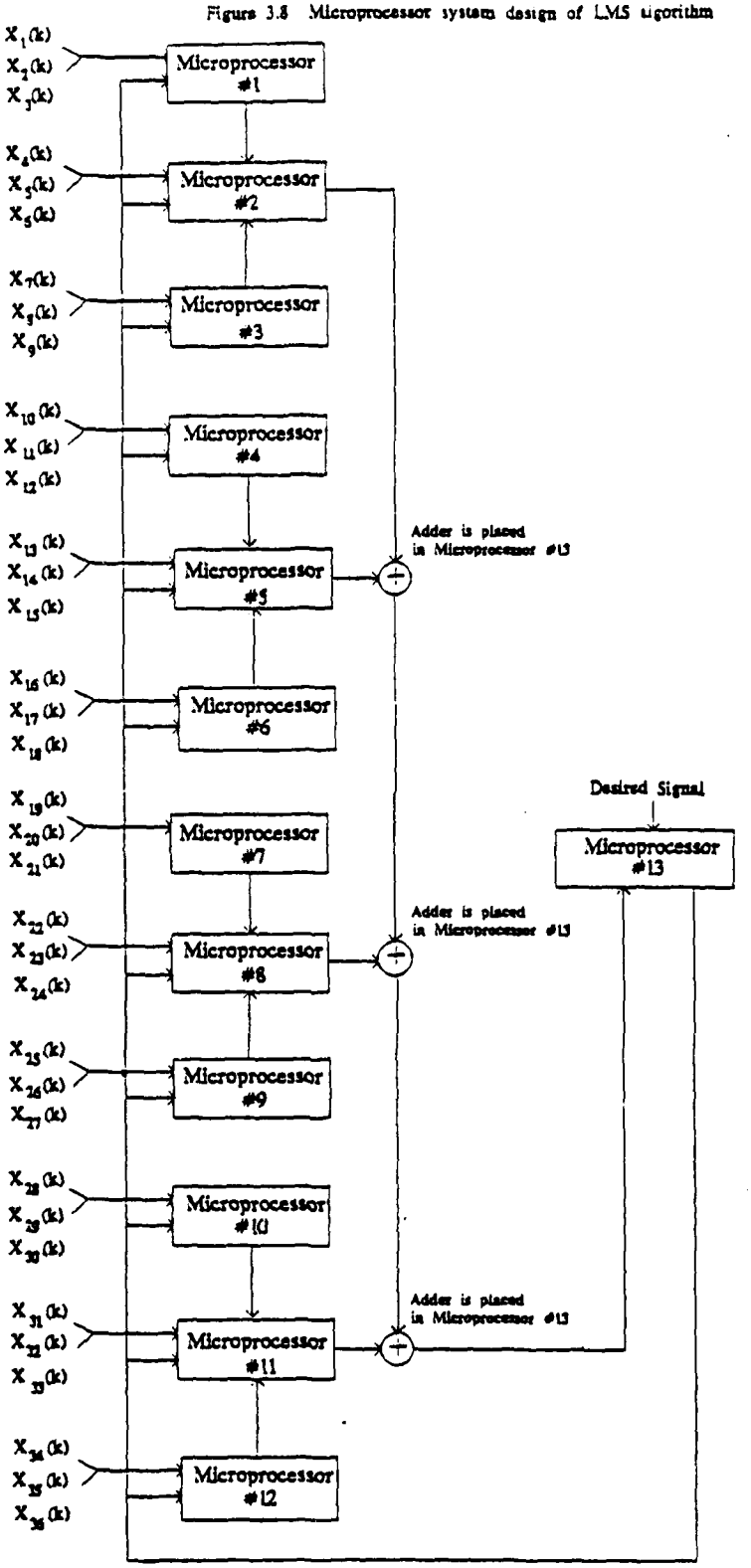
Similarly task T_2 requires 11 microprocessors and task T_3 requires 12 microprocessors. As tasks T_1 through T_3 are performed in sequence, the same 12 microprocessors can be used for the tasks leading to an initial estimate of the complexity of the LMS algorithm to be 12.

As the complexity is low, a system can be designed and its performance studied. Through further analysis it can be determined if the time involved in the communication of data between chips introduces additional computational capacity for the LMS algorithm, and thereby require more chips to implement the algorithm. As storage of intermediate results occur the memory requirement for data storage depends on the implementation.

3.3.2 Implementation considerations

It has been determined that the LMS algorithm needs 12 microprocessors. One possible arrangement of the microprocessors for implementation of the algorithm is shown in Figure 3.8. We have used 13 instead of 12 microprocessors for the ease of distribution of tasks. With the

Figure 3.8 Microprocessor system design of LMS algorithm



algorithm partitioned into various functions, the job now is to allocate the various functions among the 13 microprocessors.

Microprocessors numbered #1 through #12 perform similar operations. The additions and multiplications mentioned here are complex operations. Each complex quantity is represented as 16 + 16j i.e., 16 bits for real and 16 bits for imaginary. At the beginning the signal samples from antenna elements 1,2,and 3 are directed to processor #1, the signal samples from antenna elements 4,5, and 6 are directed to processor #2 and so on, so each processor numbered #1 through #12 gets three input samples. At the beginning, each of the processors, #1 through #12, perform three multiplication and three additions. Processor #1 performs $\sum_{i=1}^{3} W_i X_i^T$, processor #2 performs $\sum_{i=4}^{6} W_i X_i$ and so on. To obtain $Y = W X$, the output of the 12 processors are to be added, which is 11 additions. Eleven additions done sequentially takes 9.84 μs. To reduce the time needed for this computation, processor #2 adds the partial results from processor #1 and partial results form processor #3 to its partial result, to obtain

$$\sum_{i=1}^{9} W_i X_i = \sum_{i=3}^{3} W_i X_i + \sum_{i=4}^{6} W_i X_i + \sum_{i=7}^{9} W_i X_i$$

Simultaneously processors #5, #8, and #11 perform similar operations to produce partial results $\sum_{i=10}^{18} W_i X_i$, $\sum_{i=19}^{27} W_i X_i$, and $\sum_{i=28}^{36} W_i X_i$ respectively. Now processors #2,#5,#8 and #11 direct their outputs to processor #13 which adds this data to produce Y(k). This way the time spent on the 11

additions needed is only 5.64 μs.

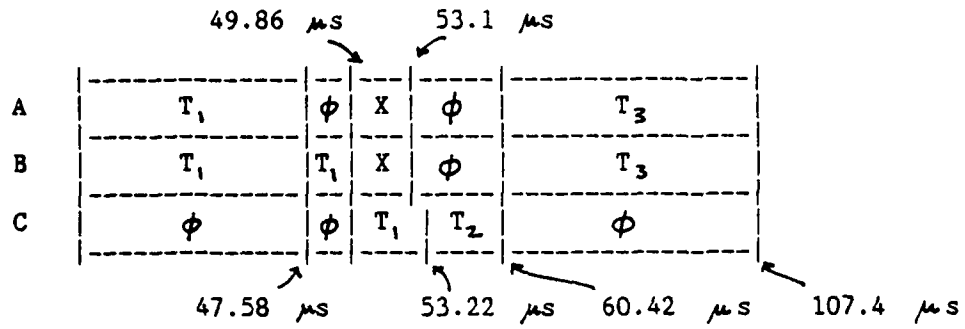
Next, Task T_2 is performed on processor #13 to obtain the error signal $e'(k)$. This signal is broadcast to processors #1 through #12. Once this is done, the 12 processors perform task T_3 to update the weights. Note that as task T_3 requires X^*, processor #1 through #12 perform the complex conjugate of X to obtain X^* while task T_2 is being performed in processor #13. Notice that equal time was required by both X^* and T_2.

When operated in this environment, the code can be written for each of the tasks and the time taken by each function is obtained. The task time scheduling [8] is shown in Figure 3.9. From this Figure the time taken for an iteration and the idle portion of the iteration time can be determined. The iteration time obtained is 107.4 μs which is in the order of the iteration time of 107.5 μs imposed by the HF constraints and the system constraints.

The time taken for each task in the system and the computational loading/chip for these tasks are given in Table 3.4. The tasks T_1 and T_3 were used to benchmark the microprocessor to have computational capacity of 0.5 MOPS. While computing task T_3 the average loading/microprocessor is 0.57 MOPS, implying that the microprocessors in the system are used efficiently. But while computing T_1, the speed of the microprocessor is only 0.41 MOPS as time is spent on passing intermediate data among the microprocessors.

The reason that the speed of microprocessor is (0.57 MOPS)

Figure 3.9 Task scheduling using microprocessor architecture for LMS algorithm



ϕ : represents idle

A : for processors #1, #3, #4, #6, #7, #9, #10, #12

B : for processors #2, #5, #8, #11

C : for processor #13

Table 3.4 Timing summary of microprocessor architecture implementation of LMS algorithm

Tasks	Execution Time (μs)	Computational loading/chip
Task T_1	53.22	0.41
Task T_2	7.2	0.55
Task T_3	46.98	0.57

Table 3.5 Processor Utilization for the microprocessor architecture consideration of LMS algorithm

Processors	Processor utilization	Idle Time (μs)
#1, #3, #4, #6 #7, #9, #10 #12	91 %	9.6
#2,#5,#8 & #11	93 %	7.32
#13	10 %	96.84

more than the capacity obtained earlier (0.5 MOPS) is that advantage is taken of the availability of 256-byte on chip instruction cache in Mc68020. Since most of the code in LMS algorithm can be written as loops, time is spent in fetching the necessary instructions only once as from the on-chip cache. This leads to savings in processor cycles which increases the speed of the microprocessor. The timing summary takes this savings into account.

The chip utilization [8] in the 13 microprocessor system can be determined. Table 3.5 shows the utilization or fraction busy time of a processor in the system. The amount of time a processor is idle per iteration is also indicated in this table, and is determined from Figure 3.9. Using the 13 processors the mean utilization of the overall system is found as

$$\frac{(107.4)13 - (9.6)8 - (7.32)4 - 96.84}{(107.4)13} = 85.5\%$$

which indicates that microprocessor system designed for LMS algorithm has high utilization of the microprocessors used.

The microprocessor design requires 98 words of storage where each word (except μ') corresponds to 32-bits of real and 32-bits of imaginary. X and W each require N words of storage and d(k) requires one word of storage, and the rest are for intermediate data storage. As each word corresponds to 8 bytes of memory, the microprocessor system requires 780 bytes of data memory.

An evaluation of the LMS algorithm using general purpose

microprocessor architecture has now been completed. An assessment of the complexity of the system implementing the algorithm was discussed and it was determined that the complexity of the algorithm using microprocessor architecture is 12. A system architecture was then developed for this algorithm using 13 microprocessors and iteration time of 107.4 s was obtained. The architecture allows the system to perform 931 iterations during 100ms and required 780 bytes of data storage.

3.4 DSP Architecture

In this section, the digital signal processor (DSP) architecture is evaluated for the LMS algorithm. The DSP chosen for this analysis was LM32900 [3] which has a cycle time of 100ns.

3.4.1 Initial Assessment on the Complexity

The complexity is given as the number of DSP chips and takes into account the following issues.

1) the computations the algorithm requires per iteration
2) time allocated for one iteration
3) the characteristics of the DSP architecture

3.4.1.1 Execution time budgets

A summary of the DSP analysis is given in Table 3.6. The first four columns of Table 3.6 are the same as that of Table 3.3. The time taken to perform an addition is the same as the

Table 3.6 Assessment on complexity of LMS algorithm using DSP architecture

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Effective number of operations	(6) Time Budget (μs)	(7) Computational loading (MOPS)	(8) Number of chips
Task T_1	286	144	142	286	50.1	5.7	1
Task T_2	4	2	2	4	0.7	5.7	1
Task T_3	324	144	180	324	56.7	5.7	1

time needed to perform a multiplication for DSP chips. The effective number of operations column (column 5 of Table 3.6) is then simply the sum of the number of additions and multiplications needed for that particular task. The procedure for allocating time budgets is similar to that done for microprocessor architecture consideration and is indicated in column 6 of Table 3.6.

3.4.1.2 Computational loading

The computational loading, as previously shown is obtained by dividing the unweighted number of operations (entries in column 2 of Table 3.6) by the execution time budget. For example, the computational loading demanded by task T_1 is

$$286/50.1\,\mu s = 5.7 \text{ MOPS}$$

The computational loading demanded by other functions is determined in the similar way and is indicated in column 7 of Table 3.6.

The speed of the DSP is determined by the time required to perform the 'operation mix' of the LMS algorithm. As stated, the operation mix constitutes the operations needed to perform one complex multiplication and complex addition. The DSP performs the needed 8 operations (4 multiplications and 4 additions) for a complex multiplication and a complex addition in 1.4μs. This results in a speed of the DSP of 5.7 MOPS. The program written for this application is in straight line coding. If the code for the 'operation mix' is written in loop

form, then the DSP operates at a speed much lower than 4.2 MOPS. This results because loops introduce additional overhead by way of Test/Branch instructions where as straight line coding does not. A disadvantage with straight line coding is that it uses more instruction memory space. On the otherhand, it reduces the complexity by increasing the speed of operation. The advantage with loop coding is that less memory space is needed but the complexity increases. The number of DSP chips needed for this application when using loop coding is 2. The LM32900 can address 64k of instruction memory. As the straight line coding for LMS algorithm does not consume so much memory the straight line coding method is adopted here. The resulting estimate on the number of DSP's needed is one with straight line coding.

As the complexity is low, a system design can be developed. By writing the software we can determine if the time per iteration obtained is within 107.5 μs.

3.4.2 Implementation considerations

As only one DSP chip is needed for implementation, the input samples from all the N antenna elements are directed to the DSP chip. The operation of the LMS algorithm is obvious in this environment. The algorithm need not be partitioned as all the tasks involved operate in the same DSP chip. The DSP chip operates task T_1, T_2 and T_3 in succession. The timing resulted from this design is indicated in Table 3.7. The overall time required for one iteration is 105.8 μs, which is

less than the constraint of 107.5μs. The loading of the chip while performing tasks T_1 and T_3 is 5.68 MOPS and 5.97 MOPS respectively.

The utilization factor is 100% as only one DSP chip is used. Note that, unlike in microprocessor design, the time needed to perform X^* is also included in iteration time, whereas in microprocessor design, the time spent to perform X^* is absorbed by task T_2.

The memory requirement for data is 2N+1 complex words. X and W each need N complex words of storage and d requires one complex word storage. As each complex word requires 4 bytes this design requires 294 bytes of data storage (2 bytes is added for storage of a real word).

In this section the suitability of the DSP architecture to the LMS algorithm has been analyzed. The initial assessment of the complexity of the algorithm using DSP architecture was found to be one. As the complexity is low we proposed a system design for 36 antenna elements. The system design completed an iteration in 105.8μs which resulted in 945 iterations in 100ms. The design required 294 bytes of data storage.

3.5 VLSI architecture

Design consideration for the VLSI computing structure was considered next. The VLSI computing structures considered are both Systolic and Wavefront [6],[10]. A key attribute of VLSI

Table 3.7 Timing summary of DSP architecture implementation of LMS algorithm

Tasks	Execution Time (μs)	Computational loading/chip
Task T_1	50.3	5.68
Task T_2	1.3	3.0
Task T_3	54.2	5.97

computing structures is their suitability to implement 'compute-bound' rather than 'input/output bound' computations. The compute bound computations in the LMS algorithm need to be identified first. The tasks are examined to determine the nature of VLSI suitability.

a) $Y = W^{T} X(k)$

This task is an inner vector-vector multiplication and the result is a scalar. As can be seen, this operation needs O(N) multiply-add steps whereas it requires O(N) input/output elements. The ability to access data from memory repeatedly in the computing structure is a consideration when the order is the same. The reason for the improved performance of compute-bound operations using Systolic or Wavefront computing arrays is the ability of repeated use of input data accessed. Once an input data is accessed from memory by the array, it is used on many processors of the Systolic or Wavefront array. The input data spends more time on computations rather than in accessing of input data. This advantage cannot be used in input-output bound operations such as this task, because a data once brought from memory and operated in one processor of the Systolic/Wavefront array has no more use. For example, when the first element of 'W' and the first element of X are brought from storage and multiplied, these data cannot be used again. Therefore this task is input-output bound and is unsuitable for VLSI computing structures.

b) $B = e' X^{*}(k)$

This task is scalar-vector product and the result is a vector. Again this task is input-output bound because the total number of input and output elements is 2N+1, whereas the task needs N multiplications. This makes it unsuitable for VLSI computing structures.

c) $W = W(k) + B$

This computation is a vector-vector addition and the result is also a vector. The total number of input-output elements needed is 3N and the total number of computations needed is N adds. This task is input-output bound which makes it unsuitable for VLSI computing structures.

The summary of VLSI suitability is provided in Table 3.8. Since there are no compute-bound operations it can be concluded that VLSI computing structures are unsuitable for LMS algorithm.

3.6 Chapter summary

An analysis was performed on the hardware realization of the LMS algorithm. Used were general purpose microprocessor architecture, digital signal processor architecture and VLSI architecture. The LMS algorithm was operating in a DSSS system using BPSK modulation scheme with a bit rate of 300 bits/sec and a processing gain of 31.

The DSP architecture was determined to be the best suited in the HF environment. The complexity is much lower than using

Table 3.8 List of compute-bound and Input-output bound functions belonging to LMS algorithm

Functions	Type of operation	Number of computations	Number of input-output elements	Operation bound
Y	inner vector product	O(N)	O(N)	input-output
B	scalar-vector product	N	2N+1	input-output
W	vector-vector addition	N	3N	input-output

the microprocessor architecture. The VLSI computing structures were unsuitable for the reasons noted.

A few observations should be noted regarding the microprocessor design and the DSP design of LMS algorithm. Table 3.9 and Table 3.10 provide the execution times required for the LMS algorithm in the microprocessor system design, and the DSP system design, respectively. The execution times were broken down into transfer instructions, arithmetic instructions, branch instructions and miscellaneous instructions. Note that a larger portion of time was spent on multiply instructions by the microprocessor system (= 73.92 μs) than the time spent on multiply operations by the DSP system (= 25.4 μs). There are two reasons for this. First, the microprocessor system does 32-bit multiplication whereas the DSP system does 16-bit multiplication. Secondly, in DSP chip, the execution of successive multiply-accumulate instructions allows parallel operations of multiplier and accumulator (DSP architecture is optimized to multiply-add operation). This allows a new multiply-accumulate instruction to be executed every cycle which saves time on multiply instructions.

Another observation shown in Table 3.10 is that the add/subtract instruction consumes a larger portion of the total execution time (27.6%) in the DSP system design, than in the microprocessor system design which is 15.5%. The reason is

that in any isolated addition required by the algorithm, the DSP chip consumes the same order of time as that of a multiplication, whereas the microprocessor consumes only 1/6 of the time required for the multiplication.

Table 3.9 Execution times required for LMS algorithm using microprocessor architecture

Operation	Time(μs)	Percent
transfer from memory	9.12	8.5
transfer reg to reg	3.36	3.1
transfer to memory	1.8	1.7
add/subtract	16.68	15.5
multiply	73.92	68.8
test/branch	2.52	2.4
Total	107.4	100

Table 3.10 Execution times required for LMS algorithm using DSP architecture

Operation	Time(μs)	Percent
transfer from memory	0.8	0.76
transfer reg to reg	28	26.46
transfer to memory	11.2	10.59
add/subtract	29.2	27.6
multiply	25.2	24
miscellaneous	11.2	10.59
Total	105.8	100

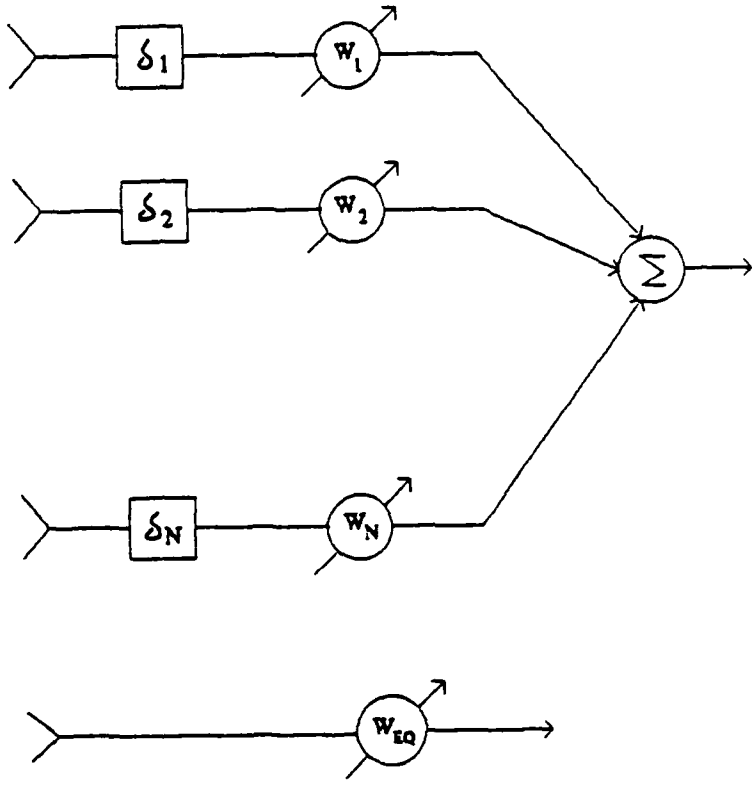
4.0 HARDWARE ARCHITECTURES FOR CONSTRAINED LMS ALGORITHM

This chapter discusses various hardware architecture implementations of the constrained LMS (c-LMS) algorithm, the feasibility of these architectures, and recommends suitable architectures. A brief introduction to the c-LMS algorithm and the equations that are to be solved by the hardware implementation are given in section 4.1. In section 4.2, an initial loading analysis procedure is derived for the c-LMS algorithm. The loading analysis provides a method to determine an initial estimate on the computational loading required by the different functions that constitute the algorithm. The loading analysis is performed for various hardware architectures once the initial estimation is determined. The hardware architectures considered were the general purpose microprocessor, the digital signal processor and the VLSI architecture. They are discussed in sections 4.3, 4.4 and 4.5 respectively. In section 4.6 a summary discusses the feasibility of these architectures and a recommended architecture for the HF application.

4.1 c-LMS algorithm

The c-LMS algorithm [18] like the LMS algorithm uses a gradient approach to obtain the optimum weights. This algorithm requires that the arrival angle of the desired signal be known apriori. The adaptive array system model, which will be used to explain the operation of the c-LMS algorithm, was presented in [18] and is shown in Figure 4.1. Although the hardware implementation deals with narrowband signals, the original

Figure 4.1 Narrowband Signal-aligned array system



broadband processor model will be used in this discussion. The model consists of N elements and J taps per element. Narrowband signals are used and a simplified model results. This model will be described later. Also shown in Figure 4.1 is an "equivalent processor" which aids in the understanding of how the c-LMS operates.

From Figure 4.1, it is evident that the c-LMS processor contains a component known as a spatial correction filter. This component performs a task that is regarded as preprocessing. The spatial correction filter guarantees that the communication signal component is identical at each element output. The delays can be calculated from the array geometry and the arrival angle of the desired signal.

From the desired signal's point, the processors in Figure 4.1 are equivalent. Each adaptive weight in the equivalent processor is equal to the sum of the weights in the vertical column above it. With these values, the signal components at the respective processor outputs are identical. By assigning a value to these equivalent weights, a desired frequency response in the look direction is selected. This process introduces J constraint conditions. Since there are N x J - J degrees of freedom this can be used to minimize the non-look direction noise power. Minimizing non-look direction noise power is equivalent to minimizing total output power. Regardless of how the weights are adjusted the constraints guarantee that the response in the look direction will not be degraded. The equation for the optimum constrained weight solution will be

presented next.

The expected value of the array output power is given by

$$E[y^2(k)] = E[\underline{W}^T \underline{X}(k) \underline{X}^T(k) \underline{W}] = \underline{W}^T \underline{R}_{xx} \underline{W} \qquad (4.1)$$

Define a J-dimensional vector that guarantees the desired frequency response and represents the summed weight values of the j vertical columns as

$$\underline{f} = \begin{bmatrix} f_1 \\ f_2 \\ . \\ f_N \end{bmatrix} \qquad (4.2)$$

The weights in the jth vertical column must sum to the selected number f_j. This constraint condition can be expressed as

$$\underline{C}_j^T \underline{W} = f_j \quad , j = 1,2, \ldots N \qquad (4.3)$$

where C_j is a NJ-dimensional vector consisting of all zero and N ones given by

$$C_j^T \; [00 \ldots 0 \ldots 00 \ldots 0 \ldots 11 \ldots 1 \ldots 00 \ldots 0 \ldots 00 \ldots 0] \qquad (4.4)$$
$$\qquad\quad N \qquad\qquad N \qquad\qquad N \qquad\qquad N \qquad\qquad N$$

A constraint matrix can then be defined that satisfies all j equations given by (4.3) as

$$\underline{C} = [C_1 \; C_2 \; .... \; C_j] \qquad (4.5)$$

The full set of constraints can then be written as

$$\underline{C}^{T}\ \underline{W} = \underline{f}$$

Although it seems like a complicated process, the constraint matrix C guarantees that the sum of the weights in the vertical columns is equal to the weights in the equivalent processor. The constrained optimization problem statement can now be formulated. The array output power, $\underline{W}^{T}\ \underline{\underline{R}}_{xx}\ \underline{W}$, must be minimized subject to the constraint condition $\underline{C}^{T}\ \underline{W} = \underline{f}$.

The optimum weight vector is found by using the techniques of LaGrange multipliers. A cost function, similar in purpose to the MSE function of the LMS algorithm, is formed by concatenating the constraint equation with a J-dimensional vector of undetermined LaGrange multipliers $\underline{\lambda}$. This cost function is then minimized with respect to the weights, i.e.,

$$\text{Cost}(W) = 1/2\ \underline{W}^{T}\ \underline{\underline{R}}_{xx}\ \underline{W} + \underline{\lambda}\ [\ \underline{C}^{T}\ \underline{W} - \underline{f}\ ] \qquad (4.7)$$

(a factor of 1/2 is added to simplify the arithmetic)

Once again, notice that the cost function is a quadratic function of the weights. It is known that the gradient of this function is zero at the minimum point. The optimum weights are then found by finding the gradient of the function and setting it equal to zero.

The gradient of the cost function is found by differentiating with respect to the weights.

$$\nabla_{cost} = \underline{\underline{R}}_{xx}\ \underline{W} + \underline{C}\ \underline{\lambda} \qquad (4.8)$$

Setting this result equal to zero yields the optimal weight

solution.

$$\underset{-xx}{R}\ \underset{-}{W} + \underset{-}{C}\ \underset{-}{\lambda} = 0 \qquad (4.9)$$

$$\underset{opt}{W} = -\underset{-xx}{R^{-1}}\ \underset{-}{C}\ \underset{-}{\lambda}$$

The LaGrange multipliers are found by realizing that the optimal weight solution must satisfy the constraint condition.

$$\underset{-}{C}^{T}\ \underset{opt}{W} = \underset{-}{f} = \underset{-}{C}^{T}\ [\ -\underset{-xx}{R^{-1}}\ \underset{-}{C}\ \underset{-}{\lambda}\ ] \qquad (4.10)$$

$$\underset{-}{\lambda} = -\ [\ \underset{-}{C}^{T}\ \underset{-xx}{R^{-1}}\ \underset{-}{C}\ ]^{-1}\ \underset{-}{f}$$

The optimum constrained weight vector can now be expressed as

$$\underset{-opt}{W} = \underset{-xx}{R^{-1}}\ \underset{-}{C}\ [\ \underset{-}{C}^{T}\ \underset{-xx}{R^{-1}}\ \underset{-}{C}\ ]^{-1}\ \underset{-}{f} \qquad (4.11)$$

As in the LMS algorithm, this algorithm uses the Method of Steepest Descent. This method states that the new weight vector is equal to the previous weight vector plus a change proportional to the negative gradient.

$$W(k+1) = w(k) - \mu \nabla_{cost}$$

In this case, the weight update equation is

$$W(k+1) = W(k) - \mu [\ \underset{-xx}{R}\ \underset{-}{W(k)} + \underset{-}{C}\ \underset{-}{\lambda(k)}\ ] \qquad (4.12)$$

The updated weight must satisfy the constraint condition, and is written as

$$\underset{-}{f} = \underset{-}{C}^{T}\ \underset{-}{W(k+1)} = \underset{-}{C}^{T}\ [W(k) - \mu [\underset{-xx}{R}\ \underset{-}{W(k)} + \underset{-}{C}\ \underset{-}{\lambda(k)}]\ ] \qquad (4.13)$$

The LaGrange multipliers, $\lambda(k)$, are then given by

$$\underline{\lambda}(k) = -[\underline{C}^T \underline{C}]^{-1} \underline{C}^T \underline{R}_{xx} \underline{W}(k) - 1/\mu\, [\underline{C}^T \underline{C}]^{-1} [\underline{f} - \underline{C}^T \underline{W}(k)] \quad (4.14)$$

and the iterative relation for the update equation is expressed as

$$\underline{W}(k+1) = \underline{W}(k) - \mu [\underline{I} - \underline{C} (\underline{C}^T \underline{C})^{-1} \underline{C}^T] \underline{R}_{xx} \underline{W}(k) + \underline{C} (\underline{C}^T \underline{C})^{-1} [\underline{f} - \underline{C}^T \underline{W}(k)] \quad (4.15)$$

For the sake of convenience, two definitions are made. Define

$$\underline{\beta} = \underline{C} (\underline{C}^T \underline{C})^{-1} \underline{f} \quad (4.16)$$

and P matrix as

$$\underline{P} = \underline{I} - \underline{C} (\underline{C}^T \underline{C})^{-1} \underline{C}^T \quad (4.17)$$

where I is the identity matrix. The update equation can then be rewritten as

$$\underline{W}(k+1) = \underline{P} [\underline{W}(k) - \mu \underline{R}_{xx} \underline{W}(k)] + \underline{\beta}$$

The covariance matrix Rxx is unknown, however, so an approximation of Rxx at the kth iteration, $\underline{X}(k)\underline{X}^T(k)$, is used. Recognizing the fact that $\underline{X}^T(k)\, \underline{W}(k) = y(k)$, the final update equation becomes

$$\underline{W}(k+1) = \underline{P} [\underline{W}(k) - \mu\, y(k)\, \underline{X}(k)] + \underline{\beta}$$

The constrained LMS algorithm requires a spatial correction filter to compensate for the misalignment of the sensor elements. A method proposed by Takao et. al., [19] merges the misalignment compensation and the weight computation into a single process. The direction of arrival of the communication

signal is used to generate a directional constraint to govern the weights.

The directional constraint is provided as

$$\underline{C}^{T} = (e^{j\Omega_1}, e^{j\Omega_2}, \ldots\ldots, e^{j\Omega_N})$$

where Ω_i is the phase of the desired signal at sensor element 'i'. We can then rewrite the expressions for the P matrix and β vector using this constraint equation

$$\underline{P} = \underline{I} - (\underline{C}^{*}\ \underline{C}^{T})/N$$

$$\underline{\beta} = \underline{C}/N$$

and the weight update equation is

$$\underline{W}(k+1) = \underline{P}\ [\underline{W}(k) - \mu y(k)\ \underline{X}^{*}(k)] + \underline{\beta}$$

From this discussion it can be noticed that P and β can be calculated from the directional information. Thus the calculation of the P matrix and β vector is not included in the hardware implementing the c-LMS algorithm. These quantities can be found prior to the first iteration of the algorithm and they do not change later.

4.2 Loading analysis of the c-LMS algorithm

The loading analysis [14] is now performed for the c-LMS algorithm.

4.2.1 Input requirements

The c-LMS algorithm involves solving the equations to update the weights. The equations are broken down into manageable functions as indicated in Figure 4.2.

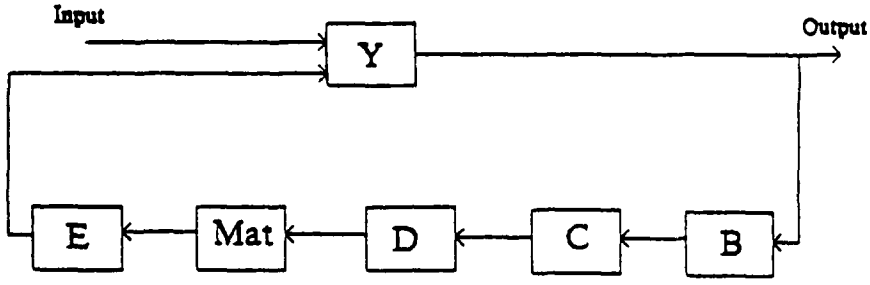
Note that the quantities P and β depend only on the steering delays and the number of antenna elements. These quantities can be calculated from directional information and knowledge of the number of antenna elements present in the array. Thus P and β remain fixed. The c-LMS algorithm considers these quantities as constant coefficients and are present in memory before the first iteration of the algorithm begins.

As can be seen from Figure 4.2, the functions into which the algorithm has been partitioned can only be operated in sequence. The next iteration of the algorithm cannot be performed until the weights have been updated by the previous iteration. Thus, the algorithm has an input-output latency and time must be determined to allocate to this latency. The average number of iterations of the algorithm needed for one convergence has been determined by simulation studies [1] to be 1155. Using the HF channel constraint, that the signals can be considered stationary for 100ms, imposes the condition that the set of optimum weights should be obtained in the period 100ms/1155 = 86 μs. Thus the input signals arrive at intervals of 86 μs.

4.2.2 Execution time budgets

A portion of this iteration time (86μs) is to be allocated

Figure 4.2 Partitioning of c-LMS algorithm for hardware realization



$$Y = W^T X$$

$$B = \mu Y$$

$$C = B X^*$$

$$D = W - C$$

$$Mat = P D$$

$$E = Mat + \beta$$

to each function of the algorithm. To allocate the execution time budget for each function, the number of real operations required by each function must be known. An analysis is done for 36 antenna elements. Note that an add operation is equivalent to a subtract operation in this analysis, so all subtract operations are referred as add operations.

1) $Y = W^{T} X(k)$

This function involves 36 complex multiplications and 35 complex additions and thus requires (36 * 6) + (35 * 2) = 286 real operations which comprises 144 real multiplies and 142 real adds.

2) $B = \mu Y$

This function requires 2 real multiplications as μ is a real quantity.

3) $C = B X^{*}(k)$

In addition to the 36 complex multiplications this function requires 36 complex subtractions to obtain X^{*}. In total this function requires (36 * 6) + 36 = 252 real operations, 144 real multiplies and 108 real adds.

4) D = W - C

This function requires 36 complex subtractions and therefore requires (36 * 2) = 72 real adds.

5) Mat = P D

This function requires 36^{2} complex multiplications and 36 * 35 complex additions. This function requires (36 * 6) + (36^{2} * 35 * 2) = 10296 real operations which comprises

5184 real multiplies 5112 real adds.

6) E = Mat + β

This function requires 36 complex additions which comprises 36 * 2 = 72 real adds.

These observations are tabulated in Table 4.1. The Table 4.1 also shows that the c-LMS algorithm requires $N^2 + 2N$ complex multiplications and $N^2 + 3N$ complex additions. Once the number of computations required by each function is determined, then we have to allocate time budget to each function. Time budgets will be discussed later for the various hardware architectures considered.

4.2.3 Computational loading

When the time budget is known the computational loading offered by each function is easily determined. Computational loading is obtained by dividing the number of real operations required for the function by the time budget allocated to that function.

4.2.4 Memory requirements

An initial estimate on the memory requirement of the c-LMS algorithm can be determined as follows. The algorithm requires $N^2 + 3N$ complex words of storage. X, W and β each require N complex words of storage and P requires N^2 complex words of storage. Because storage of intermediate results are necessary, the memory requirement varies with hardware architecture

Table 4.1 Computational complexity of c-LMS algorithm

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Complex Multiply	(6) Complex Add
Y	286	144	142	N	N
B	2	2	-	-	-
C	252	144	108	N	-
D	72	-	72	-	N
Mat	10296	5184	5184	N^2	N^2
E	72	-	72	-	N

considered.

4.3 Microprocessor Architecture Implementation

The evaluation of general purpose microprocessor architecture is now considered for the c-LMS algorithm. The microprocessor used is a Mc 68020 [2], state-of-the-art 32-bit microprocessor, which has a cycle time of 60 ns.

4.3.1 Initial Assessment on the Complexity

The following issues were considered in determining the complexity of the hardware implementing the algorithm:

1) the number of operations the algorithm needs for an iteration -- computational bounds
2) the time allocated for the iteration -- input requirements
3) the hardware architecture implementing the algorithm -- architecture considerations

The complexity of the system implementing the algorithm is expressed in terms of the number of chips needed by the system.

4.3.1.1 Execution Time budgets

How the time budget is allocated to each function is explained now. Due to the microprocessor, architecture factors that play a role in determining the complexity of the system are:

1) the ratio of the time needed for a multiply operation and for an add operation

2) the speed of the microprocessor with regard to c-LMS algorithm given in millions of operations per second.

The summary of the following discussion is shown in Table 4.2. The first column of Table 4.2 contains partitioning of the algorithm. It can be seen that a few functions are coupled as a single task. Note that functions C and D are coupled together as one task because these two functions can be performed on an element by element basis. That is, instead of computing C for all 36 elements and then computing D for the 36 elements, the two operations can be interleaved. For the same reason, functions Mat and E are combined as one task.

The second column of Table 4.2 shows the number of real operations each of the tasks need . Column 3 and column 4 of Table 4.2 indicate the number of real multiplies and real adds each of the tasks require . The microprocessor chosen has a multiplication to addition time ratio of 6 : 1. Thus the 'effective number of operations' column is obtained by performing (number of real multiplies) * 6 + (number of adds) * 1 for each task. When implemented in the microprocessor the effective number of operations for each task represents the task in terms of adds. This shows the true complexity of the task.

A portion of the sampling time (86 μs) is allocated to each task according to its 'effective number of operations'. Thus the 'time budget' column in Table 4.2 is obtained for each task by

Table 4.2 Assessment on complexity of c-LMS algorithm using microprocessor architecture

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Effective number of operations	(6) Time Budget (μs)	(7) Computational loading (MOPS)	(8) Number of chips
Task T_1	286	144	142	1006	2.26	126.55	254
Task T_2	2	2	-	12	0.03	66.7	134
Task T_3	324	144	180	1044	2.3	140.3	281
Task T_4	10368	5184	5184	36288	81 4	127.4	255

T_1 — $Y = W^T X$

T_2 — $B = \mu Y$

T_3 —
$$\begin{cases} C = B X^* \\ D = W - C \end{cases}$$

T_4 —
$$\begin{cases} E = Mat + \beta \\ Mat = P D \end{cases}$$

$$\frac{\text{(effective number of operations of that task)} * 86\mu s}{\text{(total number of 'effective number number of operations' for the algorithm)}}$$

For example, for task T_1 the time budget is obtained by

$$\frac{(1006) * 86\mu s}{(38350)} = 2.26\ \mu s$$

The execution time budgets for the other tasks are indicated in column 6 of Table 4.2.

4.3.1.2 Computational loading

The entries in the computational loading column of Table 4.2 are obtained by dividing the unweighted number of operations (entries in column 1) by the ' time budget' for that task. For example, for task T_1 the estimated computational loading is obtained as:

$$286/2.26\mu s = 126.55 \text{ MOPS.}$$

To determine the number of chips needed for each task, we have to obtain the speed of one microprocessor chip. It can be observed from Table 4.2 that Task T_4 consumes the major portion of the iteration time. The basic operation needed to perform this task is a 'complex multiply and complex add'. This can be seen in the operation Mat performed for the 'i' th element. The 'i'th row of Mat is multiplied by column vector D. Thirty-six complex multiplications and additions result, so the basic operation is repeated 36 times. Thus, the speed of the microprocessor will depend on the time the microprocessor takes to complete this basic operation. In the

worst case the microprocessor performs this operation at a speed of 0.5 MOPS.

Using these computations, the number of chips for a task can be determined. The number of chips for task T_1 is obtained as:

$$126.55/0.5 \approx 254 \text{ chips.}$$

Task T_2 needs 134 chips, task T_3 needs 281 chips and task T_4 needs 255 chips. Note that as these tasks are operated in sequence, the c-LMS algorithm needs 281 chips.

As stated previously the complexity of the architecture is determined by the number of microprocessor chips required by the algorithm. Using the general purpose microprocessor architecture the complexity of the c-LMS algorithm is 281. In summary the complexity of the c-LMS using microprocessor architecture is very high. To distribute the above tasks among 281 chips and to co-ordinate the movement of intermediate results leads to a very complex system. Data communication among 281 chips will also introduce data overhead which was not included in the execution time budgets. More general more than 281 microprocessors will be needed to bring an iteration time to 86 μs because this overhead consumes extra time. For these reasons, the microprocessor architecture is not suitable for the c-LMS algorithm.

4.4 DSP architecture implementation

In this section, the digital signal processor (DSP) architecture is evaluated for the c-LMS algorithm. The DSP

chosen for this analysis is the LM32900 [3] which has a cycle time of 100ns.

4.4.1 Initial Assessment on the Complexity

An initial assessment on the complexity of the algorithm employing DSP architecture is performed as follows.

4.4.1.1 Execution Time budgets

Table 4.3 shows the summary of the following discussion. The first four columns are filled in as they are in Table 4.2. The time taken to perform an addition and a multiplication on a DSP chip is the same. So the 'effective number of operations' column (column 5 of Table 4.3) is the sum of additions and multiplications needed by that particular task.

The procedure for determining time budgets for DSP architecture is the same as 4.3.1.1 and is indicated in Table 4.3.

It can be seen that the time budget for the various tasks obtained using microprocessor and the DSP, is more or less the same though the characteristics of the two architectures are different. A task comprising of only addition/subtraction will be allocated lesser time in the microprocessor implementation than in the DSP implementation. This results because in the DSP chip an isolated addition or a subtraction consumes the same order of time as a multiplication. For example, if the operation D = W - C (this operation has only

Table 4.3 Assessment on complexity of c-LMS algorithm using DSP architecture

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Effective number of operations	(6) Time Budget (μs)	(7) Computational loading (MOPS)	(8) Number of chips
Task T_1	286	144	142	286	2.24	127.7	23
Task T_2	2	2	-	2	0.016	127	23
Task T_3	324	144	180	324	2.54	127.55	23
Task T_4	10368	5184	5184	10368	81.2	127.7	23

T_1 — $Y = W^T X$

T_2 — $B = \mu Y$

T_3 —
$$\begin{cases} C = B X^* \\ D = W \cdot C \end{cases}$$

T_4 —
$$\begin{cases} E = Mat + \beta \\ Mat = P \cdot D \end{cases}$$

subtractions) were not included in task T_3, i.e., operations C = B X^* and D = W - C were performed in sequence, the time budgets allocated to operation D in the two implementations considered would differ. The time budget calculation for the D operation will allocate 0.84 μs in the DSP architecture but only 0.0027 μs in the microprocessor architecture.

4.4.1.2 Computational loading

The 'computational loading' column is shown in Table 4.3. The entries in this column are obtained by dividing the unweighted number of operations (entries in column 1) by the 'estimated time' allocated to that task. One interesting aspect that can be noted is that the DSP architecture provides equal loading to all the tasks involved in the algorithm, whereas the microprocessor architecture provides unequal loading among the tasks.

As noted previously the speed of the DSP chip with regard to the c-LMS algorithm can be determined by the time it takes to compute a 'complex multiply and add '. The DSP chip needs 1.4 μs to complete the required 8 operations, hence the speed of the DSP chip is 5.7 MOPS.

Now the number of chips (hence complexity of system) needed by each task can be determined. The number of chips needed for task T_1 is 127.7/5.7 = 23 chips. The tasks T_2 through T_4, all need 23 chips. Note that these tasks operate in sequence, and the c-LMS algorithm needs 23 chips. The complexity of the system is 23 DSP chips for the c-LMS algorithm.

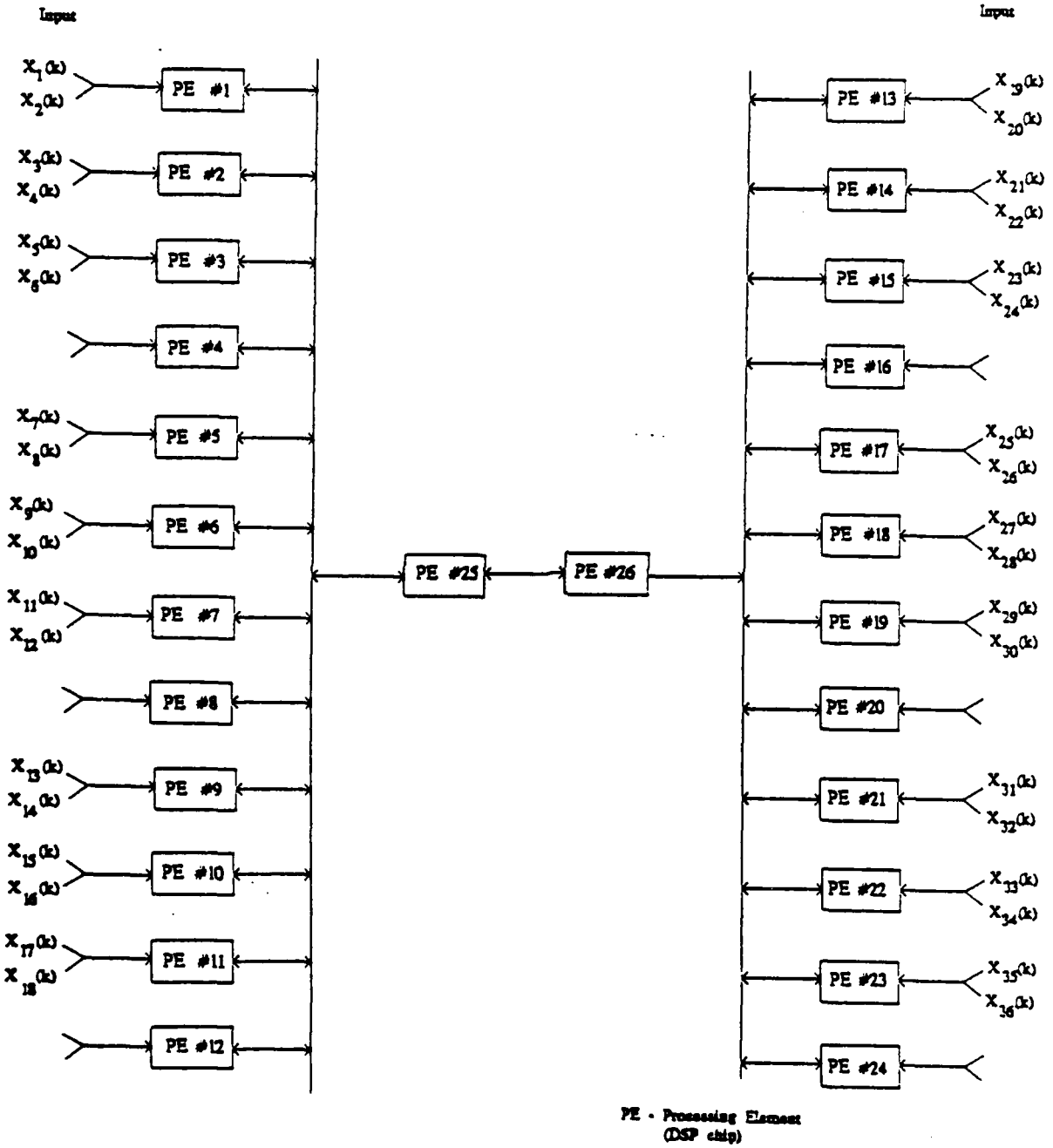
Because the complexity of the DSP architecture is moderate, a system design can be studied. Through an analysis, it can be determined if the time involved in the communication of data between chips introduces additional computational capacity for the c-LMS algorithm. Additional computational capacity would require more chips.

4.4.2 Implementation considerations

One possible arrangement of the 23 chips is shown in Figure 4.3 (DSP chips are numbered from #1 to #26). Twenty-six DSP chips have been used instead of 23 for ease of distributing the tasks. Previously the algorithm was divided into various tasks, and now the various tasks must be allocated among the 26 DSP chips.

Each complex quantity is represented as 16 + 16j, i.e., 16 bits for both real and imaginary parts. DSP chips numbered #1 through #24 (except DSP chips numbered #4, #8, #12, #16, #20 and #24) are grouped as set #1 and the DSP chips numbered #4, #8, #12, #16, #20 and #24 grouped as set #2. Initially 18 DSP chips (belonging to set #1) are provided a pair of input signal samples from the 36 antenna elements. Each of these 18 DSP chips perform $\sum_{i=1}^{2} W_i X_{Ti}$, two complex multiplication additions. To obtain $Y = W X$ (task T_1) the results are added from these 18 DSP chips. This job is assigned to chips #25 and #26. DSP chip #25 performs additions of the results from chips

Figure 4.3 DSP architecture system design of the c-LMS algorithm

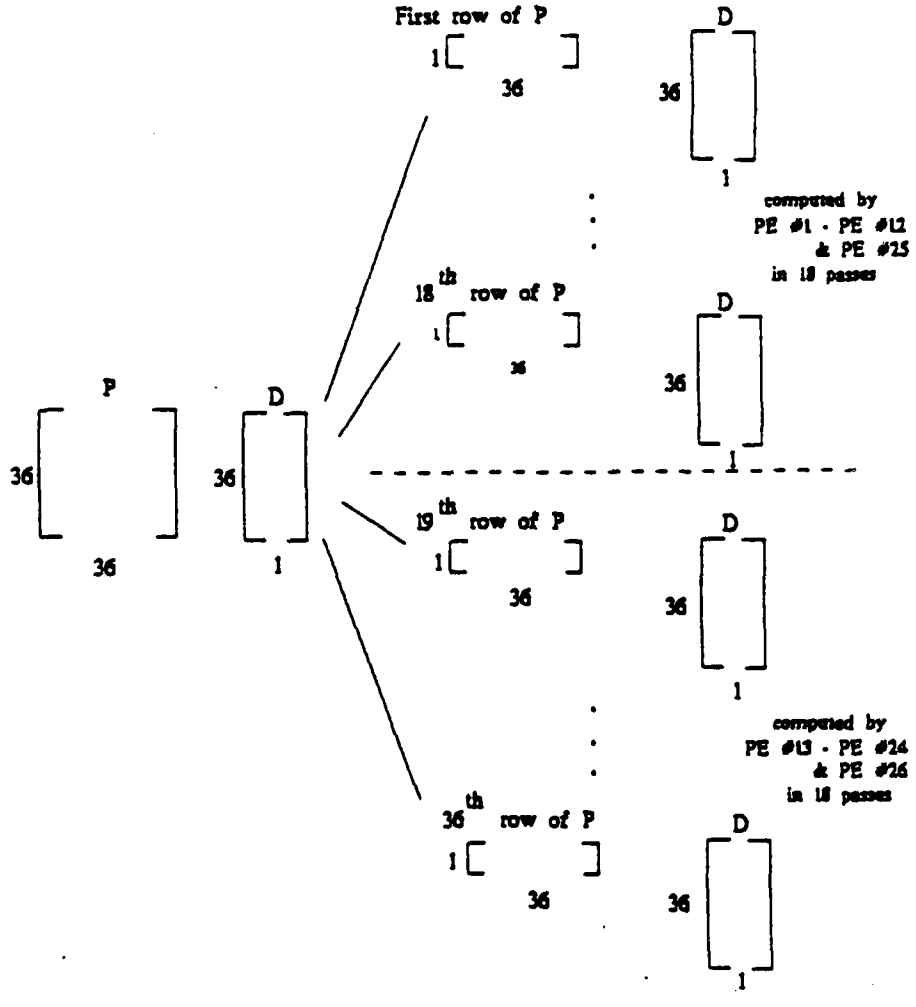


PE - Processing Element (DSP chip)

numbered #1 through #12 (except #4, #8 and #12) to obtain $\sum_{i=1}^{18}$ $W_i X_i$. Similarly chip #26 performs additions of results from chips numbered #13 through #24 (except #16, #20 and #24). Note that when the results from chip #25 and chip #26 are added, $Y = W^T X$. Thus chips #25 and chip #26 exchange data and add to obtain Y. Meanwhile DSP chips belonging to set #1 perform the complex conjugate of X (input signal samples). Next each DSP chip (#25 and #26) performs μY to obtain B. DSP chip #25 broadcasts B to chips #1 through #12 (except #4,#8 and #12), and DSP chip #26 broadcasts B to chips #13 through #24 (except #16, 20 and 24). Now DSP chips belonging to set #1 have the necessary data to begin task T_3. After the completion of task T_3 each of the DSP chips belonging to set #1 provides 2 complex entries of the vector D (this vector has in total 36 complex entries).

At this instant tasks T_1, T_2 and T_3 are completed and the remaining task to be performed is T_4. This is the most computationally intensive task of the algorithm. Up to this point the DSP chips belonging to set #2 were idle. Now these chips compute T_4. The Mat operation is a matrix-vector multiplication and is partitioned as shown in Figure 4.4. The reason for partitioning the Mat operation in this manner will be discussed later. For the Mat function, DSP chips numbered #1 through #12 and DSP chip #25 are assigned to perform one half of the computations involved (648 complex multiplications additions). DSP chips numbered #13 though #24

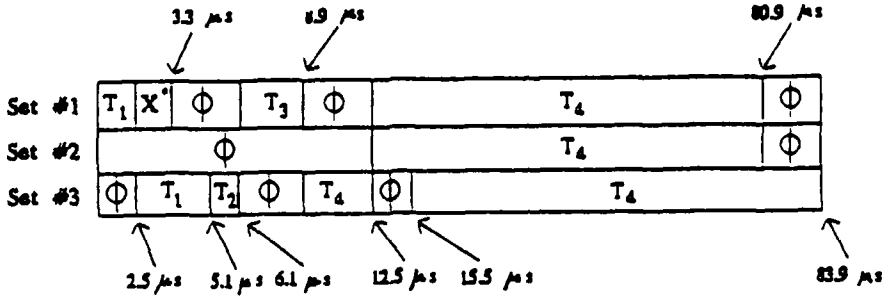
Figure 4.4 Partitioning of the 'Mat' function of the c-LMS algorithm to be computed in DSP architecture system



and DSP chip #26 perform the other half of the computations. The task T_4 is performed in a pipeline manner. Initially DSP chips #1 through #12 perform the 36 complex multiplications additions involved in the multiplication of one row of Mat and D vector (each DSP chip performs 3 complex multiplication additions). The partial results are provided to DSP chip #25 which performs the 13 complex additions (including the addition of β) to obtain the first element of the weight vector. Meanwhile DSP chips #1 through #12 perform the next set of 36 complex multiplication additions involved for the next row of Mat. Thus after 18 such passes the task T_4 is completed as DSP chips numbered #13 to #24 and chip #26 perform similar operations.

The task time scheduling [8] is shown in Figure 4.5. From this Figure the time taken for an iteration can be determined as well as the portion of the iteration time the DSP chips in the system are idle. The iteration time obtained was 83.9 μs. This time was less than the iteration time of 86 μs imposed by the HF constraints. The reason for pipelining 36 complex multiplication-additions (performed by 12 DSP chips) and 13 complex additions (performed by one DSP chip) is that each of the 12 chips requires 3 complex multiplication additions which needs 3.8 μs. The 13 complex additions performed by a single chip needs 3μs. As the times required by the two operations (3 complex multiplication-addition and 13 complex additions) are of the same order, the pipelining will

Figure 4.5 Task scheduling using DSP architecture for c-LMS algorithm



Φ - indicates that the processor is idle

set #1 - includes processors #1-#3, #5-#7, #9-#11 #13-#15, #17-#19, #21-#23

set #2 - includes processors #4, #8, #12, #16, #20, #24

set #3 - includes processors #25 & #26

be effective in reducing the waiting period. As determined earlier, task T_4 requires 18 passes leading to 71.4 μs.

The time taken for each task in the system and the computational loading/chip for these tasks are given in Table 4.4. Though the DSP chip is tuned to the operations (complex multiply and add) performed by the computationally intensive task T_4, the loading per chip was only 5.3 MOPS instead of the full capacity of 5.7 MOPS. This resulted because time was consumed in distributing the D vector entries to the various DSP chips involved in computation. While performing task T_3, the loading/chip is at full capacity of 5.7 MOPS as the computations involved are complex multiply-add. The c-LMS implementation leads to a tightly coupled system. This created communication overhead and more than the 23 chips were needed to bring the iteration time within 86 μs.

For the system using 26 chips, the processor utilization [8] can be determined. Table 4.5 shows the utilization or fraction busy time and the idle time that each DSP chip used in the system. The overall processor utilization of the 26 chip system is found to be

$$\frac{26(83.9) - 18(9.2) - 6(15.3) - 2(22.5)}{26(83.9)} = 86\ \%$$

The processor utilization is high enough to conclude that the DSP chips are effectively used.

The memory needed for data storage by the c-LMS algorithm,

Table 4.4 Timing summary of DSP architecture implementation of c-LMS algorithm

Tasks	Execution Time (As)	Computational loading/chip
Task T_1	5.1	3.11
Task T_2	1.0	2.0
Task T_3	2.8	5.7
Task T_4	75.0	5.3

Table 4.5 Processor Utilization for the DSP architecture consideration of c-LMS algorithm

Processors	Processor utilization	Idle Time (μs)
set #1	89 %	9.2
set #2	82 %	15.3
set #3	73 %	22.5

set #1 - includes processors #1-#3, #5-#7, #9-#11 #13-#15, #17-#19, #21-#23

set #2 - includes processors #4, #8, #12, #16, #20, #24

set #3 - includes processors #25 & #26

implemented in the system, is 6100 bytes (note that each complex word is represented as 16 bits real and 16 bits imaginary). P needs N^2 complex words of storage, X, β and W each needs N complex words of storage. As complex words requires 4 bytes of storage, the data require $(N^2 + 3N) * 4 = (36^2 + 3 * 36) * 4 = 5616$ bytes. The remaining 484 bytes are needed for intermediate result storage.

The c-LMS algorithm using DSP architecture has now been evaluated. The complexity of the system implementing the algorithm was assessed and found to be 23. A system architecture was developed for this algorithm using 26 DSP chips, and the iteration time was derived as 83.9 μs. The system performs 1194 iterations during 100 ms and required 5.5 Kbytes of data storage.

4.5 VLSI architecture

In this section VLSI architecture is evaluated for the c-LMS algorithm. Here a different approach than that followed for the microprocessor architecture and the DSP architecture is used. This is because not all functions can be implemented on the VLSI computing structure. Section 4.5.1 identifies the functions that can be implemented, and it is observed that only Mat function can be implemented on VLSI computing structures. Section 4.5.2 discusses issues regarding the architectures considered for the Mat function and discusses why Systolic architecture [6] is better suited for the Mat function than the Wavefront architecture [10]. Presented are here two Systolic

array designs which possess all the properties of VLSI computing structures. The functions of processor that are to be custom made and are involved in the Systolic array are determined. Also the issue of whether the Systolic architecture can be implemented, is evaluated taking into account the present technology. These issues considered for the two designs are presented in sections 4.5.3 and 4.5.4. Finally, the VLSI computing structure must be incorporated into a larger system and the various important parameters of this system are discussed in section 4.5.5. A summary of the the discussion on VLSI architecture consideration for the c-LMS is provided in section 4.5.6.

4.5.1 Algorithm considerations

To determine if the algorithm is suitable for VLSI computation, we have to first identify the compute bound operations in the c-LMS algorithm.

The following analysis holds for any number of antenna elements (N). From Table 4.6, we can infer that the Mat function is the only compute bound operation that can be implemented on a VLSI computing structure. This is because the elements of D once accessed can be used for each of the row of P and D multiplication, i.e., D is used to multiply the first row of P and D and D is used to multiply the second row of P and D and so on. Thus, this operation can be considered to be compute-bound and therefore can be implemented in VLSI computing structures.

Table 4.6 List of compute-bound and Input-output bound functions belonging to c-LMS algorithm

Functions	Type of operation	Number of computations	Number of input-output elements	Operation bound
$Y = W^T X$	inner vector product	O(N)	O(N)	input-output
$C = B X^{\beta}$	scalar-vector product	N	2N+1	input-output
$D = W - C$	vector-vector subtraction	N	3N	input-output
Mat = P D	matrix-vector product	$O(N^2)$	$O(N^2)$	compute bound
$E = Mat + \beta$	vector-vector addition	N	3N	input-output

The Mat operation is the most computational intensive of the tasks involved in the c-LMS algorithm. It required 10368 real operations and it consumed 85 % (71.4 μs of the 83.9 μs) of the iteration time for the DSP architecture implementation. By implementing this task on VLSI structure, the time spent on this task could be reduced to a large extent, thus, reducing the burden on the processor performing the input-output bound tasks of the c-LMS algorithm.

4.5.2 Architectural considerations

As noted previously the VLSI architectures considered for the c-LMS algorithm are the Systolic and the Wavefront. The primary difference between the two architectures is that the Systolic array is a synchronous computing structure whereas the Wavefront array is a self-timed computing structure. Each architecture has advantages and disadvantages so which is better depends on the task which is to be implemented. The main issues for consideration in choosing the better suited architecture are:

4.5.2.1 Speed variation

A Wavefront array enjoys a performance advantage in that results from each processor of the array are able to start computing as soon as the inputs are ready and the output is available as soon as computation is finished. Thus, if different kinds of computations are taking place in the computing array, i.e., speed variations in computations exist, then the

data-dependence property of Wavefront arrays will have an advantage. This advantage will seldom exist in regular arrays where each processor performs the same kind of computation. The task of the c-LMS algorithm implemented in a VLSI structure is the matrix-vector prcduct (Mat = P D). This task needs a regular array i.e., the same kind of computation is performed in all the processors; - multiply-add/subtract. Remember that the handshaking operation between communicating cells in a Wavefront array requires increased design complexity and hardware cost. This means from the speed variation of computations issue, Systolic arrays are better suited than Wavefront arrays to the Mat operation.

4.5.2.2 Clock skew

When different processors receive clock signals by different paths, they may not receive clocking events at the same time, potentially causing synchronization failure. These synchronization errors, due to clock skews can be avoided by lowering clock rates and/or by adding delay to the circuits, thereby slowing the computation. The Wavefront array is free from clock skew problems as it is an asynchronous system, but the Systolic array suffers from the clock skew problems. The Wavefront array implementation is advantageous only when the clock skew is high enough to degrade the speed performance of the Systolic array implementation. For example, if a processor has a clock cycle of 100ns when operating separately, but when integrated into a Systolic array had to be clocked at 150ns

(due to a clock skew of 50ns), then the speed performance is degraded. Clock skews cause problems only for very large Systolic arrays [20] and particularly in arrays which have speed variation. As will be shown later the array for this analysis has 72 processors and is a regular array (no speed variations), so clock skew is not an issue. If there are no problems due to clock skews, then Systolic arrays are better than the Wavefront arrays because of the required extra hardware for handshaking between communicating processors.

For the reasons shown, Systolic arrays are better suited for this analysis. One of the key attributes of VLSI computing structures is their simplicity and regularity [6]. Simplicity implies that the processors making up the array must be performing simple operations and regularity implies that the processors comprising the array perform similar operations. Regularity condition is crucial. In synchronous systems, such as Systolic arrays, if processors in the array perform different kinds of operations, then the array will be clocked at the rate of the slowest computation, thereby degrading speed performance.

Presented here are two Systolic array designs, A and B, which possess all the key attributes of VLSI computing structure.

4.5.3 Systolic Array - Design A

The matrix-vector product is found by repetitious multiplication of a row of P matrix with the column vector D. As

there are 36 rows in P, the basic process has to be repeated 36 times. Each entry of the matrix P and the column vector D is a complex quantity. Note that the basic operation involves 36 complex multiplications and additions. In Design A two separate linear Systolic arrays (#1 and #2) are used because complex multiplication and addition is needed to be implemented. The two Systolic arrays interact to derive the matrix-vector product. Systolic array #1 consists of processors of type-1, and Systolic array #2 consists of processors of type-2.

The type-1 processor configuration is shown in Figure 4.6(a). This processor has three input registers. These registers store the three inputs V_{in}, U_{in}, and W_{in}. When the three registers are filled with the required inputs, the processor performs either of the two operations $W_{out} = W_{in} + V_{in} U_{in}$ or $W_{out} = W_{in} - V_{in} U_{in}$ [21] (according to the control provided) i.e., either a multiply-add or multiply-subtract. Once the processor performs either of these operations the output is available on the W_{out} line. The processor also provides U_{out} which is the unchanged version of U_{in}. At every tick of the clock, the processor shifts the three inputs into the input registers; computes multiply-add or multiply-subtract; and makes the output available. This one tick of the clock is denoted as the 'unit time' of the array.

The type-2 processor configuration is shown in Figure 4.6(b). This processor also has 3 input registers but it performs only one operation multiply-add. So during a 'unit

Figure 4.6(a) Type of processor used in Systolic array #1 (Design A)

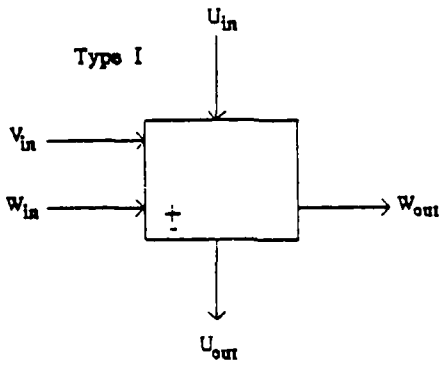
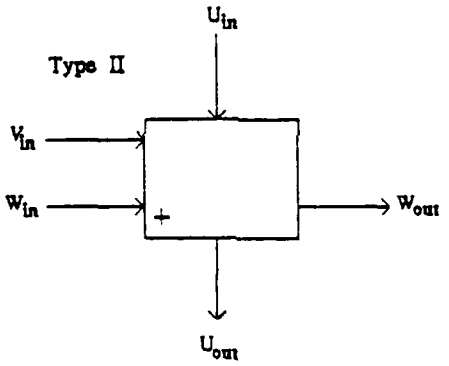


Figure 4.6(b) Type of processor used in Systolic array #2 (Design A)



time' this processor shifts the 3 inputs; computes $W_{out} = W_{in} + V_{in} U_{in}$; and makes output available.

How the two Systolic arrays interact and perform the complex matrix-vector multiplication, is now explained (Figure 4.7). To aid clarification, subscript I denotes an imaginary component and subscript R denotes a real component. The real components of the P matrix are stored in matrix P_R, and the imaginary components of the P matrix are stored in matrix P_I. P_I is supplied to array #1 and P_R is supplied to array #2. The D column vector is also separated into real D_R and imaginary D_I vectors. D_R and D_I are applied in succession to both the arrays as shown in Figure 4.8. Note that the two arrays interact and the output of array #1 is provided to array #2 and vice versa.

It is important to understand the system operation. Array #1 consists of 36 processors of type-1 arranged in a pipelined fashion. Array #2 consists of 36 processors of type-2 arranged in a pipelined fashion. As noted the corresponding processors in each array interact, i.e., the first processor of array #1 interacts with the first processor of array #2, the second processor of array #1 interacts with the second processor of array #2, and so on. In the matrix-vector product each pair of interacting processors is responsible for the 36 complex multiplications and additions (basic operation) required for the multiplication of one row of P with the column vector of D. There are 36 rows of matrix P and the 36 pairs of

Figure 4.7 Systolic array configuration to compute 'Mat' function of the c-LMS algorithm for Design A

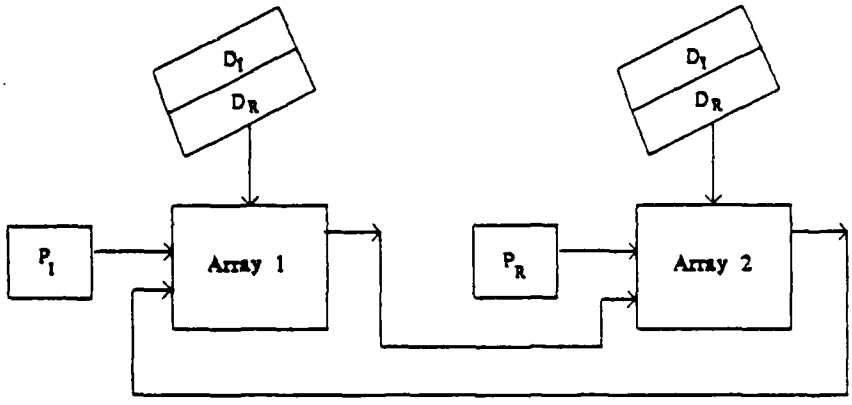
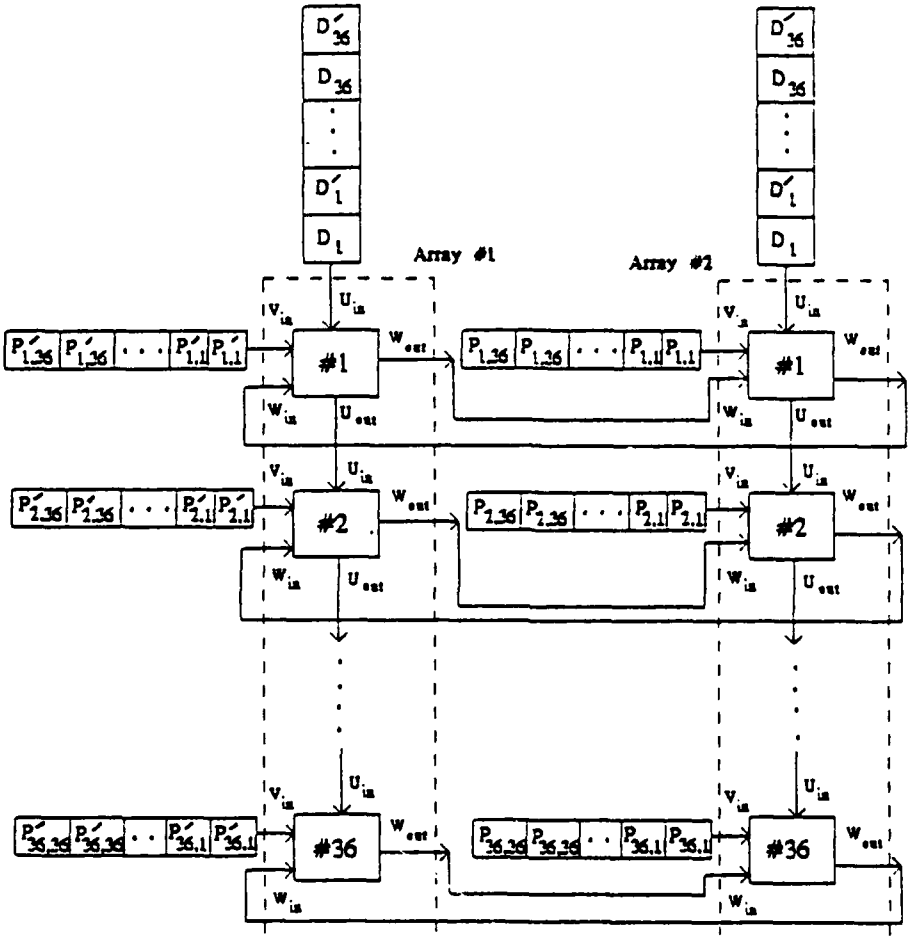


Figure 4.8 Systolic array implementation of the 'Mat' function of the c-LMS algorithm for Design A



interacting processors are sufficient to complete the matrix-vector product. For this reason, each row of the matrix P is sent to a pair of interacting processors. The real components of that row are provided to array #2 and the imaginary components of that row is provided to array #1 (Figure 4.8). To facilitate complex multiplication and addition, each element of matrix P that is provided to the processors in both arrays is repeated (Figure 4.9). For example, the real part of P_{11} is sent twice to the first processor in array #2 and the imaginary part of P_{11} is sent twice to the first processor in array #1.

The P matrix input to the processors of both the arrays #1 and #2 are provided in a skewed arrangement. The first processor of both arrays receive P matrix components during the first tick of the global clock; the second processor of both the arrays receive P matrix components only during the second tick of the global clock. To implement this arrangement, the P matrix elements provided to the second processor of the two arrays are delayed by a time unit. Correspondingly, the P matrix elements provided to the third processors of both the arrays are delayed by two time units and so on.

The complex multiplication-addition performed by the first pair of processors is explained. During the first time unit the type-1 processor of array #1 1) obtains the inputs P'_{11} ($'$ implies imaginary part of P_{11}), D_1 and W_{in} (initially this is zero); 2) computes $W_{out} = W_{in} + P'_{11} D_1$; and 3) makes

available W_{out} to the first processor of array #2, and D_1 is available to second processor of array #1. During this same time unit the first processor of array #2 1) obtains the inputs P_{11}, D_1 and W_{in} (initially this is zero); 2) computes $W_{out} = W_{in} + P_{11} D_1$; and 3) makes available W_{out} to the first processor of array #1 and D_{out} available to second processor of array #2. Thus, at the end of first time unit, $P_{11} D_1$ is available at the output of first processor of array #1 and $P_{11} D_1$ is available at the output of first processor of array #2. During the second time unit, the first processor of array #1 performs $W_{in} - P_{11}' D_1'$, (note that $W_{in} = P_{11} D_1$ is obtained from first processor in array #2), and thus obtains the real part $P_{11} D_1 - P_{11}' D_1'$ of the complex multiplication. During the same time unit the first processor in array #2 performs $W_{in} + P_{11} D_1'$, (note that $W_{in} = P_{11}' D_1$ is obtained from the first processor in array #1), and thus the imaginary part $P_{11}' D_1 + P_{11} D_1'$ of the complex multiplication is obtained. The operations performed during the next two time units will result in one complex multiplication and addition $(P_{11} + P_{11}')(D_1 + D_1') + (P_{21} + P_{21}')(D_2 + D_2')$. This is performed every two time units by the pair of processors. Note that the processors of the array #1 perform multiply-add and multiply-subtract alternatively. Because the outputs are pipelined, during a time unit, many processors of the array are active and interact in a similar manner. It can be easily seen how the matrix-vector product is obtained.

The number of time units needed by the matrix-vector

product operating in the VLSI computing structure can be computed. The time unit is the clock period of the global clock when the processors of the array are clocked. There are 36 pairs of processors which implies 36 stages of pipelining. Each pair needs to perform 36 complex multiplication-addition i.e., 72 time units, and each complex multiplication-addition consumes two time units. It can be seen that as the first row of processors finish the 36 complex multiplication-additions assigned to them, at every time unit, the successive pairs complete their assigned task. Thus after 72 time units, there is one output obtained for every time unit. Therefore the number of time units

$$= 36 + (72-1) = 107 \text{ time units.}$$

The speedup figure achieved by using this Systolic array instead of performing the Mat operation on a uniprocessor is

$$= 4 * 36^2 / 107 = 48.45$$

since matrix-vector product requires 36^2 complex multiplication additions, and each complex multiplication addition requires 4 time units in a uniprocessor.

4.5.3.1 Technology Considerations - Design A

The number of processors needed for the VLSI computing structure and the interconnections among them has been determined. Also needed is the time unit and the number of processors that can be incorporated into a chip taking into

consideration the technological constraints.

The time taken by the processor in performing the operation multiply-add or multiply-subtract determines the time unit. Assumed is the time taken by the processor to perform an addition which is the same as the time taken to perform a subtraction. Thus, the time unit is the time taken by the processor to load the inputs, perform multiply-add (and rounding), store the outputs and adjust for the clock skew. As previously determined the array does not posses severe problems due to clock skews, thus this clock skew will be a small fraction of the multiply-add or multiply-subtract time and is not an issue. A complex word has been represented as 16-bits integer for real and 16-bits integer for imaginary. Thus all multiplications and additions performed involve 16-bit operands. The present technology allows this operands shift, multiply-add and makes output available in 75 to 100ns (for example, TRW VLSI chip TDC1043 performs a 16-bit multiply-add and outputs in 100ns). For this design, the time unit chosen was 100ns which is feasible with present technology. The time unit fixes the time taken by the matrix-vector product in the VLSI structure as 107 * 100ns = 10.7 μs.

Remember that the 26-chip DSP system needed 71.4 μs to perform this matrix-vector product whereas the VLSI structure designed consumes only 10.7 μs. This is a considerable saving and lessens the burden on the hardware performing the remaining operations of the c-LMS algorithm. The complexity of the Systolic array can be determined when the number of chips needed

by array #1 and #2 is known.

First, the number of 2-input gates required by array #1 are figured. A k-bit adder/subtracter requires 20k 2-input gates and a k-bit multiplier requires $20.k^2$ 2-input gates [12]. Each processor of array #1 is of type-1 and requires one multiplier, one adder, and one subtracter; all of them perform 16-bit computations. Therefore the number of 2-input gates required by a type-1 processor is

$$= 20 . 16^2 + 20 * 16 + 20 * 16 = 5760.$$

As there are 36 type-1 processors in array #1 the gate count is

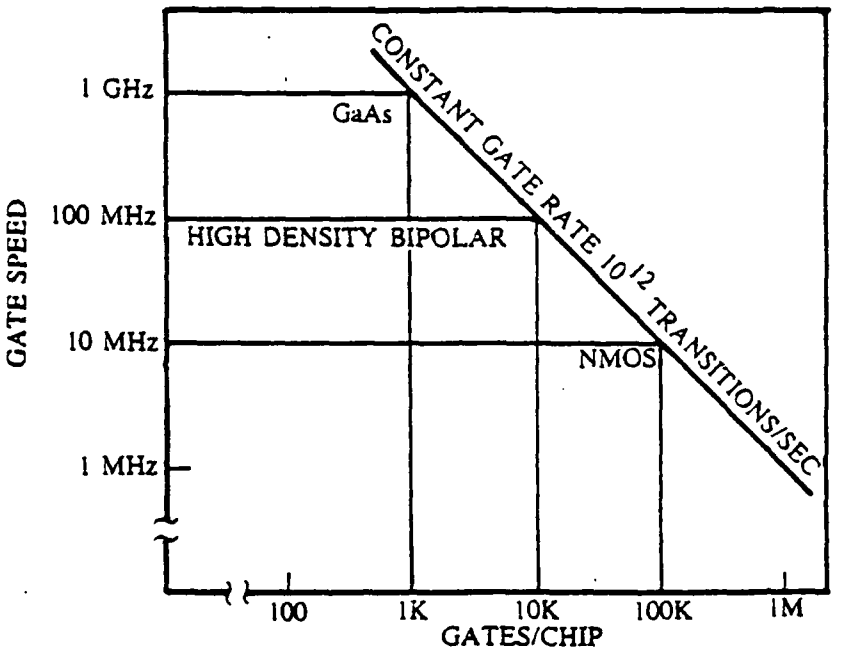
$$= 36 * 5760 = 207360.$$

The sample rate is defined as the rate at which the multiplier-adder/subtracter structure is capable of accepting data. The time unit of 100 ns implies that the processor accepts the input data every 100ns, thus, the sample rate of the processor is 10 MHz (1/100ns). The gate rate is defined as the product of the number of 2-input gates required by the array and the sample rate. The gate rate provides the measure of the amount of computation that is performed on a given chip. Figure 4.9 [22] provides the gate rate achieved by various materials. Notice that an upperlimit exists on the number of transitions of gates/sec that can occur in a chip - 10^{12} transitions/sec/chip. The gate rate achieved by array #1 is

$$= 207360 * 10 * 10^6 = 2.07 * 10^{12} \text{ transitions/sec}$$

The maximum number of transitions/sec that can occur on a

Figure 4.9 Example gate rates [22]



chip is 10^{12}, therefore array #1 needs

$$= 2.07 * 10^{12} / 10^{12} = 2.07 = 3 \text{ chips.}$$

In short 12 processors of type-1 have been placed in one chip.

By performing similar calculations, array #2 has a gate rate of $1.96 * 10^{12}$. The technology constraint of 10^{12} transitions/sec/chip results in

$$= 1.96 * 10^{12} / 10^{12} = 1.96 = 2 \text{ chips.}$$

Three chips for array #2 are used, to reduce the number of pins per chip. This implies that 12 processors of type-2 are included in one chip. A total of 6 chips, 3 chips each by Systolic array #1 and Systolic array #2 were used to implement the Mat function. The primary disadvantage of Systolic arrays is the need for large number of input-output pins [8]. The number of input-output pins needed by each chip is 608 pins. This number implies a large chip, if this is not feasible, for implementation then more of chips are needed.

4.5.4 Systolic array -Design B

Design-A calls for two one dimensional Systolic arrays, and in design B a two dimensional array is considered. Design-B requires less number of pins per chip which was the desired result. P matrix elements do not change during the iteration of the c-LMS algorithm and this advantage can be used. Instead of accessing the elements of the P matrix from the memory in Design-B, the P matrix-elements are stored in memory of each processor performing the multiply-add/subtract operation as in

[23],[24].

The Systolic array uses two types of processors, type-1 and type-2. Type-1 processor configuration is shown in Figure 4.10(a). This processor has three input latches for the inputs V_{in}, U_{in} and W_{in}. Depending on the control given the processor shifts in these inputs performs $W_{out} = W_{in} + V_{in}$ (U_{in}) or $W_{out} = W_{in} - V_{in}$ (U_{in}) and makes data available as W_{out}. Note the input (U_{in}) is not a operand but an address to the operand which resides in the processor. The processor receives the address as U_{in} and decodes it inside the processor to obtain the operand and uses this operand in the multiply-add or multiply-subtract operation. The processor also sends out U_{in} and V_{in} unchanged and is made available on the output line U_{out} and V_{out}. Thus, during a time unit, the processor shifts in the data, computes a multiply-add or a multiply-subtract and makes available the output data. Type-2 processor configuration is shown in Figure 4.10(b). This processor is similar to type-1 processor except that it can perform only operation $W_{out} = W_{in} + V_{in}$ (U_{in}) (multiply-add).

The Systolic array configuration using the two types of processors is shown in Figure 4.11. It contains 36 type-1 processors as the first column and 36 type-2 processors as the second column. Each row of processors (consisting of one type-1 processor and one type-2 processor) is responsible for 36 complex multiplication-addition. (basic operation) These are necessary for the multiplication of one row of P matrix with

Figure 4.10(a) Type of processor used in Systolic array #1 (Design B)

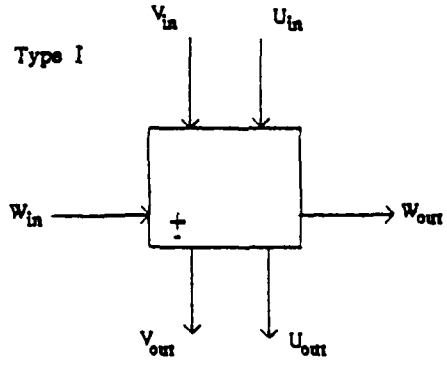


Figure 4.10(b) Type of processor used in Systolic array #2 (Design B)

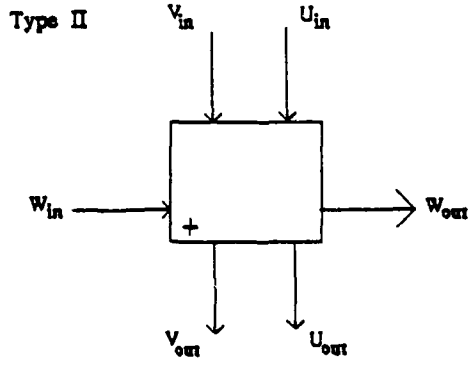
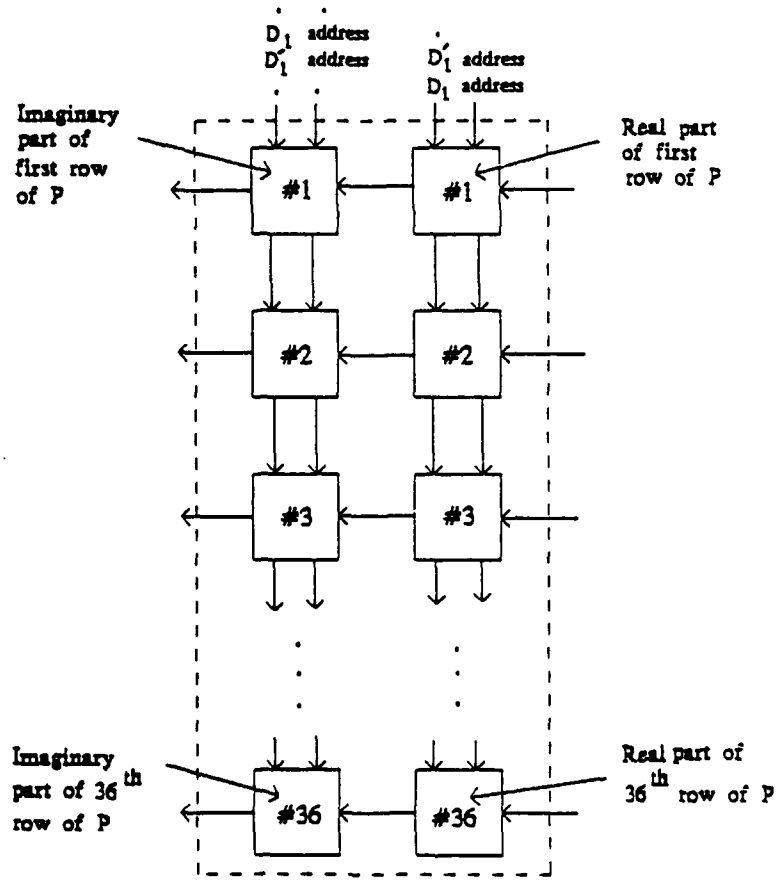


Figure 4.11 Systolic array implementation of the 'Mat' function of the c-LMS algorithm for Design B

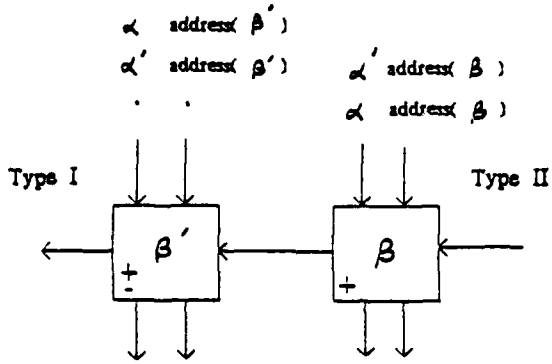


the D vector. The operation of the first row of processors in performing a complex multiplication-addition is now explained. The real components of the first row of the matrix P are stored in the type-2 processor of the first row of the Systolic array. The imaginary components of the first row of the P matrix are stored in the type-1 processor of first row of the Systolic array. Similarly, the other rows of the P matrix are stored in corresponding rows of the Systolic array.

During the first time unit, data is provided to the type-2 processor only and is delayed by a time unit to the type-1 processor. This makes the computation of the (Figure 4.12) complex multiplication-addition possible. To illustrate, consider how a complex multiplication-addition is performed by two processors. Let the complex quantities to be multiplied be $\alpha + j\alpha'$ and $\beta + j\beta'$, and their product added to $\gamma + j\gamma'$. During the first time unit, only processor type-2 is operating as type-1 processor has its inputs lagged by one time unit. During the first time unit the type-2 processor shifts in α, the address of the operand β, and γ. The type-2 processor then performs $\gamma + \alpha\beta$ and sends the addition to type-1 processor; the type-2 processor also sends α, and the address to processor in the second row.

During the second time unit, both processors of first row operate. The type-2 processor shifts in α', the address of β and γ'; computes $\gamma' + \alpha'\beta$ and makes $\gamma' + \alpha'\beta$ available to type-1 processor. During the same time unit, type-1 processor 1) shifts in α', the address of β and $\gamma + \alpha\beta$ (from processor

Figure 4.12 Complex multiplication-addition



type-2), 2) computes $\gamma + \alpha\beta - \alpha'\beta'$ (this processor performs multiply-add and multiply-subtract alternatively), and 3) stores in memory. Note that the resulting quantity is the real component of the complex multiplication-addition.

Yet to be configured is the imaginary component. During the third time unit, the type-2 processor begins the next complex multiplication addition but the type-1 processor still operates on the previous complex multiplication-addition; the imaginary component has yet to be obtained. During this time unit the type-1 processor shifts in α, the address of β', and from type-2 processor; computes $\gamma' + \alpha'\beta + \alpha\beta'$; and stores this in memory. The imaginary component is now configured.

Thirty-six operands are stored in each processor. The 36 operands in each processor have 36 different addresses. These same set of addresses are used by every other processor, which constitute the Systolic array, to store their respective 36 operands. With this arrangement the address can be pipelined through the Systolic array. It can be seen that when the D vector elements and the address of the P matrix elements are pipelined through the Systolic array, the complex multiplication addition of other rows are possible.

The number of time units needed by the matrix-vector product operating in the VLSI computing structure can now be determined. The first complex multiplication-addition takes 3 time units but subsequent complex multiplication addition requires only 2 time units. Then required are 2(36) + 1 time units for the first row of processors of the Systolic array to

complete the 36 complex multiplication-addition. The remaining 35 rows require another 35 time units to finish their operations. In total, the number of time units for the matrix-vector operation is

$$= 2(36) + 1 + 35 = 108 \text{ time units.}$$

The speedup achieved by using this Systolic array instead of operating the matrix-vector product on a uniprocessor is

$$= 4 * 36^2 / 108 = 48.$$

This result occurs because the matrix-vector product requires 36^2 complex multiplication additions and each complex multiplication addition requires 4 time units.

4.5.4.1 Technology Considerations - Design B

The number of processors needed for the VLSI computing structure has been determined for the matrix-vector product. The time unit and the number of processors need to be configured that can be incorporated into a chip realizing the technological constraints.

The time taken by the processor in performing multiply-add or multiply-subtract operation, determines the time unit. Assumed is that the time taken by the processor to perform an addition is the same as the time taken to perform a subtraction. Thus the time unit is the time 1) needed to shift in the inputs, 2) decode the address supplied to obtain the one of the operands, 3) perform multiply-add and store the outputs. A complex word has been represented as 16-bit integer for real

and 16-bit integer for imaginary. Thus, all multiplications and additions performed involve 16-bit operands. The present technology allows the operations to be performed by the processor in 200ns. This time unit of 200ns fixes the time taken by the matrix-vector product in the VLSI computing structure as

$$= 108 * 200ns = 21.4\mu s.$$

The number of chips needed by the Systolic array designed can be determined for an idea of the number of processors that can be squeezed into a single chip. A k-bit storage requires 8k 2-input gates. Each row of processors of the Systolic array requires 2 multipliers, 2 adders, a subtracter, and storage for 72 operands (36 real and 36 imaginary components of a row of P matrix). So each row required

$$= 2.20.16^2 + 3.\ 20.16 + 72\ .8.16 = 20416 \text{ gates}$$

As there are 36 rows of processors in the Systolic array, required were

$$= 36 * 20416 = 734976 \text{ gates.}$$

Not included are the gates required by the decoder and other control necessary. So the above estimate is doubled, which gives the number of 2-input gates needed by the Systolic array = 2 * 734976 = 1469952 gates.

The sample rate, the rate at which the multiplier-adder/subtracter is capable of accepting data is 5Mhz(1/200ns). The gate rate then achieved by the Systolic array design is

$$= 1469952 * 5 * 10^6 = 7.3 * 10^{12} \text{ transitions/sec.}$$

The resulting number of chips required by the Systolic array was

$$= 7.3 * 10^{12} / 10^{12} = 8 \text{ chips}$$

As there are 36 rows of the Systolic array, to distribute the computational load evenly among the chips, 9 chips for implementation were used. Implied is that 4 rows (4 processors of type-1 and 4 processors of type-2) of Systolic array are implemented on a single chip.

All data entering and leaving the chip require 16 pins each and the address of the operands residing inside the processors require 6 pins (address requires 6-bits to fetch one from 36 operands stored in each processor). The design contains 216 pins per chip (the DSP chip LM32900 has 172 pins). The chip appears feasible for implementation with present technology.

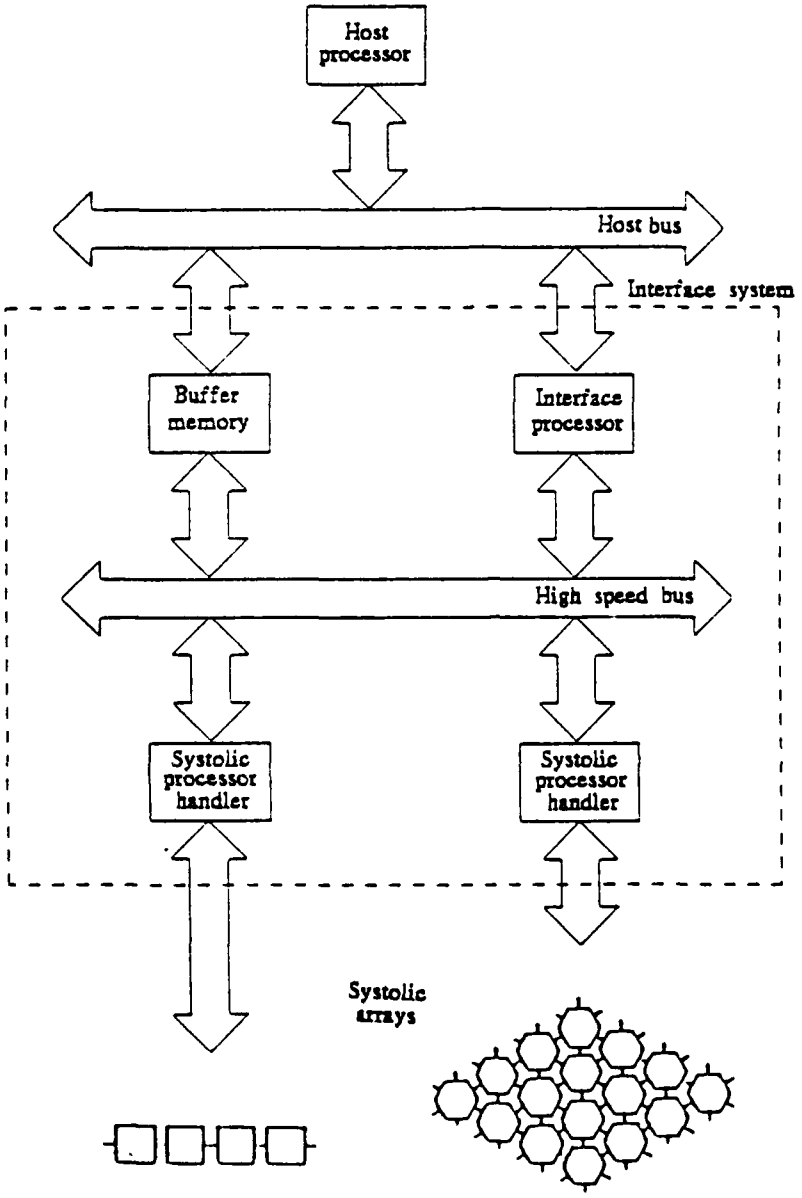
The VLSI computing structure has now been designed for the only compute-bound operation of the c-LMS algorithm, Mat =P D. Known is the time taken by this operation using the Systolic array, the number of chips needed by the array and the number of data pins needed by each chip. Integration of this Systolic array in an array processing system is the next step.

4.5.5 Interface to Array Processor Systems

The Systolic array is interfaced [12], [11] to a host as shown in Figure 4.13. The functions of each block is discussed briefly

1) Host: The host provides commands and data to the Systolic

Figure 4.13 Interface system for custom VLSI chips [12]



processor. The host does all the input-output bound operations of the c-LMS algorithm and sends data (D vector) to the Systolic processor to perform the matrix-vector product. It receives the result provided by the Systolic processor (Mat vector). The host then proceeds to complete the remaining operations of the c-LMS algorithm. Discussed later will be how a lower bound on the computational capacity of the host can be determined.

2) Interface processor: The interface processor transfers data and commands to the Systolic array from the host and transfers result from the Systolic processor to the host.

3) Systolic processor handler: This processor serves as an interface between the special high-bandwidth bus and a Systolic array. It generates address for the buffer memory accesses and run-time control signals for the Systolic array.

4) Buffer Memory: These memory units are used as buffers between the low-bandwidth host bus and the special high bandwidth buses in the interface system. By holding data that are to be used repeatedly by the Systolic arrays, the arrays can proceed with high speed without consuming much host-bus bandwidth. We can determine the storage requirement for our system.

5) High-bandwidth bus: The input/output of Systolic arrays is usually much higher than host bus because Systolic array consists of many processors requiring data at the same instant at a high rate.

The parameters that are required for the Systolic array system are the host computational capacity, the storage (buffer) requirement, and the bandwidth of the Systolic array bus.

The host functions include sending commands to the Systolic array and performing the input-output bound operations for the c-LMS algorithm. The input-output bound operations to be performed by the host are

$$Y = W^T X$$
$$B = \mu Y$$
$$C = B^* X$$
$$D = W - C$$
$$E = Mat + \beta$$

and require in total 648 real operations (290 real multiplications and 394 real adds). The time allocated to these 648 operations determine a lower bound on the computational capacity of the host.

4.5.5.1 Design A Systolic array system parameters

The time taken to perform the matrix-vector product in the Systolic array was found to be 10.7 μ s. The iteration time for the c-LMS algorithm was found to be 86 μ s. Thus the host has 75.3 μ s to perform the remaining 648 real operations. For the host this results in computational capacity of 8.6 MOPS.

The bandwidth of the Systolic bus is determined as follows. In design A, two separate interacting linear Systolic arrays are used. First the bandwidth required by array #1 needs to be determined. At the peak instant (when all the processors of array #1 are operating) this array inputs 73 words and outputs

36 words (each word is 16 bits or 2 bytes wide). The need for 73 words comes from: 1) P matrix imaginary components require 36 words (one by each processor), 2) D vector requires 1 word, and 3) the input from the other array (array #2) requires 36 words. Thirty-six words come from each processor to the array #2. Thus, every 100 ns (time unit as discussed earlier), array #1 uses the bus for 109 words thereby requiring 109 * 2/100ns = 2180 Mbytes/sec. Array #2 operates in a similar way so it also requires 2180 Mbytes/sec of bus bandwidth. As both array #1 and array #2 operate simultaneously the bus bandwidth required is 4360 Mbytes/sec. The resulting bandwidth is enormous because many parallel buses are used. The buffer capacity can now be determined. The buffer is used to store P matrix, D vector and the result Mat requiring 5328 bytes.

4.5.5.2 Systolic design B system parameters

The Systolic array in design B requires 21.6 μs to perform the matrix-vector product. This leaves 64.4 μs of the iteration time for the host to perform the remaining 648 operations. The computational capacity of host is fixed at 10.06 MOPS.

The bandwidth required for the Systolic bus will be less than that required by design A because the P matrix resides inside the Systolic processors. The Systolic array inputs 38.75 word, 36 words required as input of intermediate results, 2 words for the D vector, and 6 bits each for the 2 address buses. The array also outputs 36 words. Thus the Systolic array

uses 74.75 words every 200 ns (time unit). This means that the array requires 74.75 * 2/200 ns = 373.75 Mbytes/sec bandwidth bus. The buffer requirement is only for D and Mat vectors which requires 288 bytes of memory.

4.5.6 Summary for VLSI design

First the algorithm was examined and the operations that could be implemented on a VLSI computing structure were identified. The only compute-bound operation present was the matrix-vector product (Mat = P D) and it was considered for implementation. Two VLSI architectures, Systolic and Wavefront, were considered for the VLSI computing structure. Due to speed variations and clock skew factors the Systolic architecture is better suited to this problem.

Two Systolic array designs for the 'Mat' operation were examined. A Table comparing the parameters of the two designs is shown in Table 4.7. Each design had some advantages and disadvantages and the choice depended on the resources available.

The advantages of design A over design B were:

1) Takes less time to perform the 'Mat' operation. Design A took 10.7 μ s and Design B took 21.6 μ s. Design A reduced the burden on the Host performing the remaining operations (8.6 MOPS)
2) Required a simple chip to be fabricated. The chip performed simple functions like multiply-add and multiply-subtract. The chip in design B had to perform additional functions

Table 4.7 Systolic array system design summary for c-LMS algorithm

	Design A	Design B
Number of processors	72	72
Computational capacity/chip	20 MOPS	10 MOPS
number of chips	6	9
number of data pins/chip	608	216
Time Taken for Mat operation	10.7 μs	21.6 μs
Speedup factor	48.5	48
Host computational load	8.6 MOPS	10.06 MOPS
Systolic bus bandwidth	4360 Mbytes/sec	373.75 Mbytes/sec
Buffer memory	5328 bytes	288 bytes

such as decoding of address to obtain the operands and storing 72 bytes in each processor.

The advantages of design B over design A were:

1) Each chip required fewer pins for input/output of data than that required by each pin of design A.
2) The Systolic bus bandwidth required was far less than that required by design A, because P matrix is stored inside the processors.

The Systolic design had interrelated system parameters. To reduce the bandwidth of the Systolic bus, a slower PE was needed. This means the time taken to compute Mat operation increased which in turn, increased the computational capacity of the host.

4.6 Chapter summary

In this chapter various hardware architecture implementation were considered for the c-LMS algorithm. Evaluations of hardware architecture were performed to determine the suitability of the c-LMS algorithm for general purpose microprocessor, Digital signal processor and custom designed VLSI processors.

The general purpose architecture was found not suitable for the c-LMS algorithm. This was concluded because the general purpose microprocessor architecture's arithmetic unit was not tailored to optimize the multiply-add operation found commonly in signal processing algorithms. Too many chips were needed

(281) to implement the algorithm.

The digital signal processor architecture has its arithmetic unit tailored to perform the multiply-add operation. The DSP chip chosen for analysis performed complex multiply-add at the rate of 5.7 MOPS. This resulted in a moderate complexity of 23 for the DSP architecture implementing the c-LMS algorithm. An architecture for the c-LMS algorithm in a 26 DSP chip environment was developed and it was found that an iteration took 83.9 μs to provide 1194 iterations during the period of 100ms. This ensured, on a average, one convergence per 100ms when the HF channel was stationary.

Next two VLSI architectures, Systolic and Wavefront, were considered. Due to the lack of speed variations and clock skews, the Systolic architecture was better suited than the Wavefront architecture for the only compute bound operation 'Mat' (matrix-vector product) in the c-LMS algorithm. Two Systolic array designs were developed for the matrix-vector product. Simple functions were needed by the custom VLSI processors designed for the regular Systolic arrays. This resulted in a simple processor design. Another positive attribute was low numbered chips, 6-9 were needed to compute Mat operation. The main problems encountered in both designs were the need for large number of pins/chip, and high Systolic bus bandwidth.

These analysis led to the conclusion that the DSP architecture is best suited for the c-LMS algorithm. With the advent of more powerful DSP processors, the complexity of the DSP

architecture will be further reduced.

5.0 HARDWARE ARCHITECTURES FOR UPDATE COVARIANCE ALGORITHM

The purpose of this chapter is to discuss the various hardware architecture implementations for the Update covariance algorithm, to study the feasibility of these architectures, and to recommend suitable architectures. In section 5.1 a brief introduction to the Update covariance algorithm is presented. The algorithm that is to be implemented by the hardware is discussed. Section 5.2 presents an initial loading analysis for the Update covariance algorithm. This analysis provides a method for determining an initial estimate on the computational loading required by different functions which constitute the algorithm. The loading analysis is performed for various hardware architectures. The hardware architectures considered were the general purpose microprocessor, the digital signal processor and the VLSI architecture and are discussed in sections summary discussing the feasibility of these architectures and a recommended architecture is discussed in section 5.6.

5.1 Update covariance algorithm

Both the LMS and constrained LMS algorithms circumvent computational problems associated with the direct calculation of a set of weights by using effective estimates. The simpler calculations that result allow them to frequently update the weights in order to compensate for the time-varying environment. Recursive processors such as the Update Covariance algorithm [25], can also be used to avoid these computational difficulties. These algorithms recursively perform matrix

inversion so that direct matrix inversion is never required. Although they also avoid direct matrix inversion, recursive processors represent a significant departure from the algorithms previously discussed.

The optimum weight solution given by Weiner-Hopf can be expressed as

$$W_{-opt} = R_{-xx}^{-1} P_{-}$$

The Update Covariance algorithm estimates the sample covariance matrix rather than rely on gradient methods that asymptotically approach an optimal solution. For stationary environments these recursive procedures compute the best possible selection of weights (based on least-squares fit to the data received) at each sampling instant, while in contrast the LMS method is only asymptotically optimal.

The operation of this algorithm can be described as a series of complex computations solely intended to calculate the optimal weight solution. As the name implies, the Update Covariance algorithm uses the sample covariance estimate, R, to summarize the effect of de-emphasizing the past data. The new sample covariance matrix estimate is given by

$$R_{-xx}(k+1) = \alpha R_{-xx}(k) + X_{-}^{*}(k+1) X_{-}^{T}(k+1) \qquad (5.1)$$

The new estimate is equal to the new computed value $X^{*}(k+1) X^{T}(k+1)$ plus the past estimate scaled by a factor of α, α is a number between 0 and 1 that is used to determine the significance of past data. The inverse estimate then becomes

$$R_{-xx}^{-1}(k+1) = 1/\alpha \left[R_{-xx}^{-1}(k) + 1/\alpha\, X_{-}^{*}(k+1)\, X_{-}^{T}(k+1)\right] \qquad (5.2)$$

Note that calculating the inverse in this manner however, would require matrix inversion, which is due to its complexity is a procedure to be avoided. Therefore, it is useful to invoke the following matrix identity

$$[P^{-1} + M^{*T} Q^{-1} M]^{-1} = P - P M^{*T} [M P M^{*T} + Q]^{-1} M P \qquad (5.3)$$

This identity is applied to equation 5.2 to obtain $R_{-xx}^{-1}(k+1)$ in the form

$$R_{-xx}^{-1}(k+1) = 1/\alpha \left[R_{-xx}^{-1}(k) - \frac{R_{-xx}^{-1}(k)\, X_{-}^{*}(k+1)\, X_{-}^{T}(k+1)\, R_{-xx}^{-1}(k)}{\alpha + X_{-}^{T}(k+1)\, R_{-xx}^{-1}(k)\, X_{-}^{*}(k+1)}\right] \qquad (5.4)$$

The optimum weight solution can then be found by utilizing the Weiner-Hopf equation

$$W_{-opt} = \hat{R}_{-xx}^{-1} P_{-}$$

Multiplying both sides of equation 5.4 by the vector P yields the Update Covariance weight Update equation.

$$W_{-}(k+1) = 1/\alpha \left[W_{-}(k) - \frac{R_{-xx}^{-1}(k)\, X_{-}^{*}(k+1)\, X_{-}^{T}(k+1)\, W_{-}(k)}{\alpha + X_{-}^{T}(k+1)\, R_{-xx}^{-1}(k)\, X_{-}^{*}(k+1)}\right] \qquad (5.5)$$

Thus the hardware implementing the Update Covariance algorithm has to follow these two steps:

1. Estimate the inverse sample covariance using equation 5.4

2. Calculate weight solution using equation 5.5

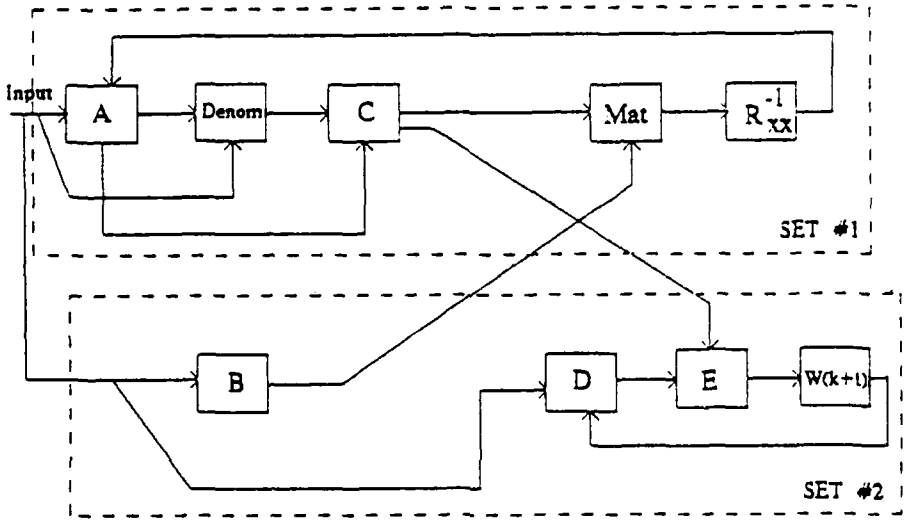
5.2 Loading Analysis

The loading analysis [14] was performed for the update covariance algorithm. This includes determining the input requirements, execution time budgets, computational loading and memory requirement for the update convariance algorithm.

5.2.1 Input requirements

The Update covariance algorithm implementation involves solving the equations to update the covariance matrix and then form the optimum weights. The algorithm are decomposed into manageable functions and are indicated in Figure 5.1. As seen in Figure 5.1, the update covariance algorithm exhibits parallelism which can be exploited to reduce the computational burden. Still during every iteration the covariance matrix, the weights have to be updated. This gives the algorithm a input-output latency and the time allocated to this latency needs to be determined. The adaptation process of obtaining a new set of 'optimum weights' and the covariance matrix was considered as one convergence. The average number of iterations that the algorithm needs for one convergence, determined by simulation studies [1], is 26. Using the HF channel constraint, that the signals can be considered stationary for 100ms, imposes the condition that the set of optimum weights and covariance matrix should be obtained during this period. This makes the iteration time for the Update covariance algorithm 3.8 ms.

Figure 5.1 Partitioning of Update covariance algorithm for hardware realization



$$A = R_{xx}^{-1}(k)\, X^{*}(k+1)$$

$$\text{Denom} = X^{T}(k+1)\, A + \alpha$$

$$C = A\ (1/\text{Denom})$$

$$\text{Mat} = C\, B$$

$$R_{xx}^{-1} = \frac{1}{\alpha}\left[R_{xx}^{-1}(k) - \text{Mat}\right]$$

$$B = X^{T}(k+1)\, R_{xx}^{-1}(k)$$

$$D = X^{T}(k+1)\, W(k)$$

$$E = C\, D$$

$$W(k+1) = \frac{1}{\alpha}\left[w(k) - E\right]$$

The input requirement of the system implementing the update covariance algorithm has now been determined with a sampling period of 1/3.8 ms. Although the algorithm requires more computations than the c-LMS algorithm, the parallelism exhibited by the update covariance algorithm and its requirement of 26 iterations for a convergence, reduces the complexity of the system.

5.2.2 Execution Time budgets

Once the input requirements are determined, the functions constituting the algorithm have to be calculated once every 3.8 ms. The time budget to be allocated to each function can be determined by finding the number of real operations required by each function. The following summarizes the requirement for each function:

1) $A = R_{-xx}^{-1}(k)\, X_{-}^{*}(k+1)$

This function requires 36^2 complex multiplications and 36 * 35 complex additions. To obtain X^{*}, 36 real subtractions are required. Therefore, this function requires $(36^2 * 6) + (36 * 35 * 2) + 36 = 10332$ real operations which is composed of 5184 real multiplies and 5148 real adds.

2) $\text{Denom} = X^{T}(k+1)\, A + \alpha$

This function requires 36 complex multiplications and 35 complex additions to perform $X^{T}(k+1)\, A$. The final addition with α requires one real add as α is a real quantity.

Therefore this function requires (36 * 6) + (35 * 2) + 1 = 287 real operations which is composed of 144 real multiplies and 143 real adds.

3) C = A/Denom

First 1/Denom is obtained. It requires 2 real multiplications, 2 real divisions (considered equivalent to multiplication), and 1 real addition; that gives a total of 5 real operations. Later 1/Denom is multiplied with A to obtain C and this operation requires 36 complex multiplications. This means this operation requires (36 * 6) + 5 = 221 real operations which is composed of 148 real multiplies and 73 real adds.

4) $B = X^{T}_{(k+1)} R^{-1}_{-xx}(k)$

This function is a vector-matrix multiplication and requires 36^2 complex multiplications and 35 * 36 complex additions. This results in $(36^2 * 6) + 2\,(36 * 35) = 10296$ real operations which is composed of 5184 real multiplies and 5112 real adds.

5) Mat = C * B

This function requires 36^2 complex multiplications which gives $(36^2 * 6) = 7776$ real operations which is composed of 5184 real multiplies and 5112 real adds.

6) $R^{-1}_{-xx} = 1/\alpha\,[R^{-1}_{-xx}(k) - Mat]$

$[R^{-1}_{-xx}(k) - Mat]$ requires 36^2 complex subtractions and later as each member of the matrix has to be multiplied with the

real quantity α, this operation requires $2 * 36^2$ real multiplies. Thus this function requires $(36^2 * 2) + (2 * 36^2) = 5184$ real operations consisting of 2592 real multiplies and 2592 real adds.

7) $D = X^T W$

This function is similar to function Y and requires 286 real operations.

8) $E = C * D$

This function requires 36 complex multiplications and in total requires (36 * 6) = 216 real operations consisting of 144 real multiplies and 72 real adds.

9) $W = 1/\alpha [W - E]$

The operation (W - E) requires 36 complex subtractions. Later each member of the vector has to be multiplied with the real quantity α ; this operation requires 2 * 36 real multiplies. This function's total is (36 * 2) + (2 * 36) = 286 real operations consisting of 144 real multiplies and 142 real adds.

These observations are tabulated in Table 5.1. The Table also indicates that the Update covariance algorithm requires $3.5 N^2 + 4.5 N$ complex multiplications and $2N^2 + 2N$ complex additions.

5.2.3 Computational loading

The computational loading is obtained by dividing the number of real operations required by the function by the time

Table 5.1 Computational complexity of Update Covariance Algorithm

Functions	Real operations	Real Multiply	Real Adds	Complex Multiply	Complex Add
A	10322	5184	5148	N^2	N^2
Denom	287	144	143	N	N
C	221	148	73	N	-
B	10296	5184	5112	N^2	N^2
Mat	7776	5184	2592	N^2	-
Rxx	5184	2592	2582	$N^2/2$	-
D	286	144	142	N	N
E	216	144	72	N	-
W	144	72	72	N /2	-

budget allocated to that function. Computational loading is discussed in later sections.

5.2.4 Memory Requirements

An initial estimate on the memory requirement of the update covariance algorithm can be obtained. The algorithm requires N^2 + 2N complex words of storage. X and W each require N complex words of storage and R_{-xx}^{-1} requires N^2 complex words of storage. Note that because storage of intermediate results are necessary the memory requirement depends on the hardware architecture considered.

5.3 Microprocessor Architecture Implementation

In this section the general purpose microprocessor architecture was evaluated for the Update covariance algorithm. The microprocessor used was MC 68020 [2], state of the art 32-bit microprocessor, which has a cycle time of 60 ns.

5.3.1 Initial Assessment on the Complexity

The complexity of the hardware implementing the algorithm is a function of

1) the number of operations the algorithm needs for an iteration, i.e., computational bounds
2) the time allocated for the iteration, i.e., input requirements
3) the architecture implementing the algorithm, i.e., architecture considerations.

When all these issues are taken into consideration, an estimate on the number of chips (microprocessors) needed by the system implementing the algorithm can be obtained. The complexity of the system implementing the algorithm is indicated as the number of chips needed by the system. The loading analysis of the algorithm in the microprocessor architecture environment can be performed.

5.3.1.1 Execution Time budgets

Factors that play a role in determining the complexity of the system due to the microprocessor architecture are:

1) the ratio between the time needed to do a multiply operation and an add operation
2) with regard to update covariance algorithm the speed of the microprocessor given in millions of operations per second.

Table 5.2 provides a summary of the following discussion. The first column shows the partitioned algorithm. The second column gives the number of real operations for each function and the third and fourth columns gives the required number of real multiplies and real additions for each function.

The microprocessor chosen has a multiplication and addition time ratio of 6 : 1. The effective number of operations column is obtained by performing (number of real multiplies) * 6 + (number of real adds) for each function. The effective number of operations for each function represents the function in terms of adds. These provide the true complexity of the

Table 5.2 Assessment on complexity of Update Covariance algorithm using microprocessor architecture

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Effective number of operations	(6) Time Budget (ms)	(7) Computational loading (MOPS)	(8) Number of chips
A	10322	5184	5148	36036	1.524	6.77	14
Denom	287	144	143	1007	0.0426	6.74	14
C	221	148	73	961	0.041	5.39	11
B	10296	5184	5112	36216	1.61	6.4	13
Mat	7776	5184	2592	33696	1.425	5.46	11
Rxx	5184	2592	2582	18144	0.7674	6.76	14
D	286	144	142	1006	0.9	0.32	1
E	216	144	72	936	0.84	0.257	1
W	144	72	72	504	0.45	0.32	1

function when implemented in the microprocessor. This is indicated in column 5 of Table 5.2.

Because the algorithm exhibits parallelism, there are two sets of processors implementing the algorithm. It can be seen from Figure 5.1 that functions A, Denom, C, Mat and Rxx operate with one set of processors (set #1) and functions B, D, E and W operate with another set of processors (set #2). Notice that for the set #1 processors to operate Mat function, they need B vector (the result of B function) from the set #2 processors. Similarly for the set #2 processors to operate function E, they need the C vector (the result of the C function) from the set #1 processors. Thus, there exists a data exchange between the two sets of processors.

One iteration of the Update covariance algorithm can be thought of as taking place in two phases. During the first phase functions A, Denom and C take place on set #1 processors, during the same instant, function B operates on set #2 processors. At the end of phase one, the two sets of processors exchange data, i.e., set #1 provides C vector to set #2 and set #2 provides B vector to set #1. During the second phase, set #1 processors operate functions Mat and Rxx. During the same instant, set #2 processors operate functions D, E and W. It can be seen from Table 5.2 that set #1 processors compute in total 89844 operations (see column 5 of Table 5.2) and set #2 processors compute in total 38662 operations. Thus, as set #1 processors compute more number of operations than those computed by set #2, the execution time budgets are dictated by the set #1

processors.

During phase one, set #1 processors compute 38004 operations (functions A, Denom and C) and during phase two, computes 51840 operations (Mat, Rxx). The total time for an iteration is 3.8 ms. The time is divided between the two phases proportional to the number of operations needed. This results in phase one assigned (38004)/89844 = 1.61 ms and phase two assigned (51840)/89844 = 2.19 ms. Once the time allocated to each phase is determined, then the execution time budgets of each function constituting the phase can be obtained. Each function of phase one is then allocated a portion of 1.61 ms. Function A is allocated (36036/38004) * 1.61 ms = 1.524 ms, function Denom is allocated (1007/38004) * 1.61 ms = .0426 ms and function C is allocated (961/38004) * 1.61 ms = 0.041 ms. Similarly functions constituting phase two are allocated a portion of time 2.19 ms. Function Mat is allocated (33696/51840) * 2.19 ms = 1.425 ms, and function Rxx is allocated (18144/51840) * 2.19 ms = 0.7674 ms.

Time has to be allocated also to the functions performed by set #2 processors. As stated earlier, the 1.61 ms is allocated to phase one. During this phase, set #2 processors operate only function B. Thus function B has a execution time budget of 1.61 ms. Allocated to phase two is 2.19 ms and during this phase, set #2 processors operate functions D, E and W, a total of 2446 operations. Thus, function D is allocated (1006/2446) * 2.19 ms = 0.9 ms, function E is allocated (936/2446) * 2.19 ms = 0.84 ms and function W is allocated (504/2446) * 2.19 ms = 0.45 ms.

5.3.1.2 Computational loading

The entries of the computational loading column of Table 5.2 is obtained by dividing the unweighted number of operations (entries in column 2) by the 'time budget' for that function. For example, for function A the estimated computational loading is obtained as

10322/1.524ms = 6.77 MOPS.

To determine the number of chips needed for each function, the speed of one chip with regard to the update covariance algorithm must be obtained. As can be seen from column two from Table 5.2, functions A and B constitute the major portion of the computations involved. The basic operation needed to perform these functions is a 'complex multiply and complex add'. The speed of the microprocessor with respect to Update covariance algorithm will thus depend on the time the microprocessor takes to complete this basic operation. The microprocessor performs this computation at a speed of 0.5 MOPS.

Using these computations the number of microprocessors needed for each function can be determined. For example, the minimum number of microprocessors needed by function A is

6.77/0.5 = 14 microprocessors.

The number of microprocessors required by other functions is indicated in the last column of Table 5.2. Remember that functions A, Denom, C, Mat and Rxx operate on set #1 processors. Which means the number of processors (or microprocessors) required by the set #1 will be determined by the function (operating on set #1) which requires the largest number of

microprocessors. It is seen that functions A, Denom and Rxx require 14 microprocessors each set #1 contains 14 microprocessors. Now we have to determine the number of processors required by set #2. The functions operated on set #2 are B, D, E and W and it can be seen that function B requires the largest number of microprocessors, which is 13. Therefore set #2 consists of 13 microprocessors. Because total microprocessors required by the Update covariance algorithms is the sum of microprocessors required by set #1 and set #2, the minimum number of microprocessors required by Update covariance algorithm is 27.

As noted previously the complexity of the architecture is determined by the number of chips required by the algorithm. The complexity of the Update covariance algorithm using the general microprocessor architecture was found to be 27. The complexity is high when compared to the following analysis of the DSP architecture. Data communication among the 27 chips introduces data overhead which was not included in the execution time budgets. It can be seen that more than 27 processors will be needed to bring the iteration time to 3.8 ms. Due to the high complexity (when compared to DSP chip) the microprocessor architecture is not considered favorable for implementation of the update covariance algorithm.

5.4 DSP architecture implementation

In this section the digital signal processors (DSP) architecture is evaluated for the update covariance algorithm.

The DSP chip chosen for this analysis was the LM32900 [3] which has a cycle time of 100ns.

5.4.1 Initial Assessment of the Complexity

An initial assessment of the complexity of the update convariance algorithm employing DSP architecture can be configured, as was done with the microprocessor architecture evaluation. The complexity is determined by the number of DSP chips and takes into account the following issues.

1) the computations the algorithm requires per iteration
2) time allocated for one iteration
3) the characteristics of the DSP architecture

5.4.1.1 Execution Time budgets

Table 5.3 gives a summary of the discussion which follows. The first four columns are filled as they were in Table 5.2. The time needed to perform an addition is the same as a multiplication for the DSP chip. This means the 'effective number of operations' column (column 5 of Table 5.3) is simply the sum of the number of additions and multiplications needed by that particular task.

The procedure in allocating the time budget is similar to that done for the microprocessor architecture. Two sets of processors, set #1 and set #2 operate in parallel to perform an iteration of update covariance algorithm in 3.8 ms. Functions A, Denom and C require 10830 operations, and functions Mat and Rxx require 12960 operations. Thus, the time allocated to phase

Table 5.3 Assessment on complexity of Update Covariance algorithm using DSP architecture

(1) Functions	(2) Real operations	(3) Real Multiply	(4) Real Adds	(5) Effective number of operations	(6) Time Budget (ms)	(7) Computational loading (MOPS)	(8) Number of chips
A	10322	5184	5148	10322	1.65	6.26	2
Denom	287	144	143	287	0.045	6.38	2
C	221	148	73	221	0.035	6.31	2
B	10296	5184	5112	10296	1.73	5.95	2
Mat	7776	5184	2592	7776	1.242	6.26	2
Rxx	5184	2592	2582	5184	0.848	6.11	2
D	286	144	142	286	0.92	0.32	1
E	216	144	72	216	0.69	0.313	1
W	144	72	72	144	0.46	0.313	1

one is (10830/23790) * 3.8 ms = 1.73 ms, and the time allocated to phase two is (12960/23790) * 3.8 ms = 2.07 ms. The execution time budgets of functions A, Denom and C are allocated a portion of 1.73 ms. Thus, function A is allocated (10322/10830) * 1.73 ms = 1.65 ms, function Denom is allocated (287/10830) * 1.73 ms = 0.045 ms, and function C is allocated (221/10830) * 1.73 ms = 0.035 ms.

Execution time budgets for functions Mat and Rxx are allocated a portion of 2.07 ms. Thus function Mat is allocated (7776/12960) * 2.07 ms = 1.242 ms, and function Rxx is allocated (5184/12960) * 2.07 ms = 0.848 ms. Because the set #2 processors perform function B during phase one function B has an execution time budget of 1.73 ms. During phase two the set #2 processors operate functions D, E and W. This means that the time budgets for function D are (286/646) * 2.07 ms = 0.92 ms, for function E are (216/646) * 2.07 ms = 0.69 ms and for function W are (144/6464) * 2.07 ms = 0.46 ms.

5.4.1.2 Computational loading

The computational loading demanded by function A is

10322/1.65ms = 6.26 MOPS.

The computational loading demanded by other functions is determined in the similar way and is indicated in column 7 of Table 5.3.

The number of chips required by the algorithm can now be determined. As noted earlier, the speed of the DSP chip can be determined by the time it takes to complete a complex multiply

and add. The DSP chip requires 1.4 s to complete the required 8 operations, that makes the speed of the DSP chip with regard to the Update covariance algorithm is 5.7 MOPS. Using this information the number of chips required for function A is 2. It has been found that set #1 processors require 2 chips and set #2 processors require 2 chips; in total the algorithm demands 4 DSP chips.

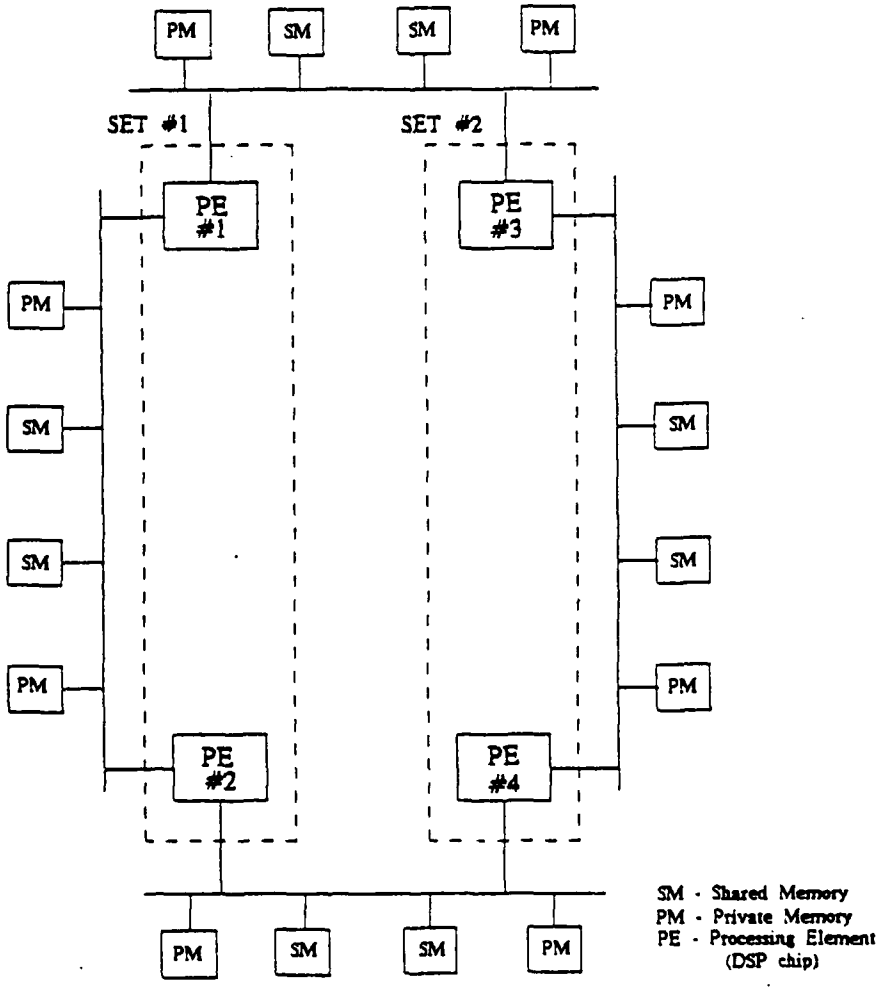
Though the number of operations involved in the update covariance algorithm is large when compared to the LMS and the c-LMS algorithm, due to parallelism present, the complexity of the algorithm using DSP architecture has been shown to be low, 4. The low complexity led to the design of a system architecture. This system architecture aids in determining whether communication overhead is present. This enables one to detemine if the complexity of the algorithm is increased by the additional computational capacity introduced by the communication overhead. The memory requirement for data storage is next determined. The data storage requirement depends on the system architecture designed.

5.4.2 Implementation considerations

It has been determined that the update covariance algorithm needs 4 DSP chips for a design using 36 antenna elements. One possible arrangement of the DSP chips is shown in Figure 5.2. With the algorithm partitioned into various functions, the job now is to allocate the various functions among the 4 DSP chips.

The partitioning and allocation of the algorithm in this

Figure 5.2 DSP implementation of the Update Covariance algorithm



environment is shown in Figure 5.3. Each complex quantity is represented as 16 + 16j i.e., 16 bits for real and 16 bits for imaginary. To start the algorithm a copy of the input samples X was provided to each DSP chip. Note that DSP chips #1 and #2 belong to set #1 and DSP chips #3 and #4 belong to set #2. The functions A, Denom and C were each broken down into two equal parts as shown in Figure 5.3 and DSP chips #1 and #2 compute each part. If necessary DSP chips #1 and #2 exchange data to compute a function. During this same period while chips #1 and chips #2 are computing, functions A, Denom and C, chips #3 and #4 compute function B. Because two chips are allocated to this function, the computational loading is divided equally among the two chips.(Figure 5.3) Remember the complete B vector is obtained only when the intermediate results obtained from chip #3 and chip #4 is added. This operation is performed by chip #3 and chip #4, this means both chips now posses a copy of the B vector. Phase one operations are now over.

The two sets of processors now exchange the data, C vector and B vector. No CPU cycles are lost in this data exchange as the data that are to be exchanged are placed in common memory to both the sets of processors. The phase two operations now begin. Chips #1 and #2 perform Mat and Rxx, and the computations are divided equally among the two chips. (Figure 5.3) At the instant the covariance matrix is updated, chips #3 and #4 perform the functions D, E and W to update the weights. The functions are divided among the two chips as shown in Figure 5.3.

Figure 5.3 Partitioning of the functions of the update covariance algorithm to DSP chips

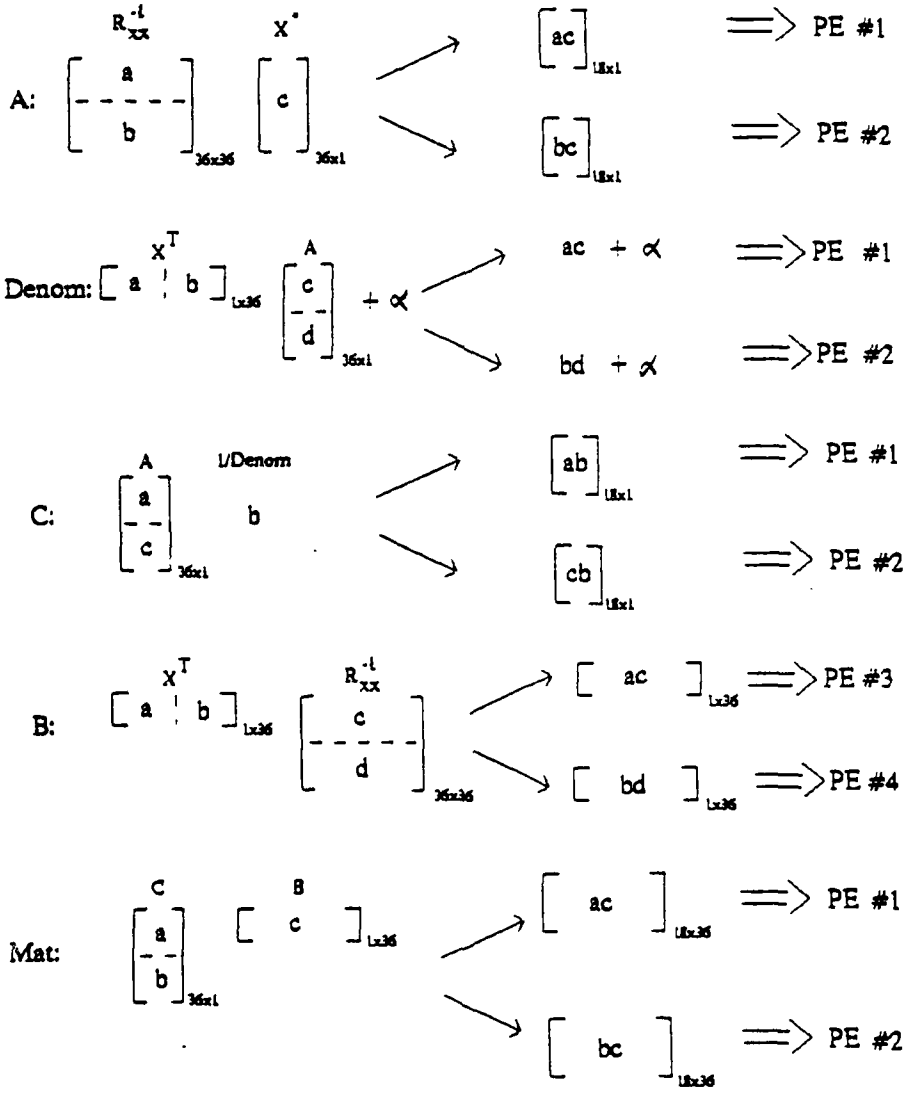
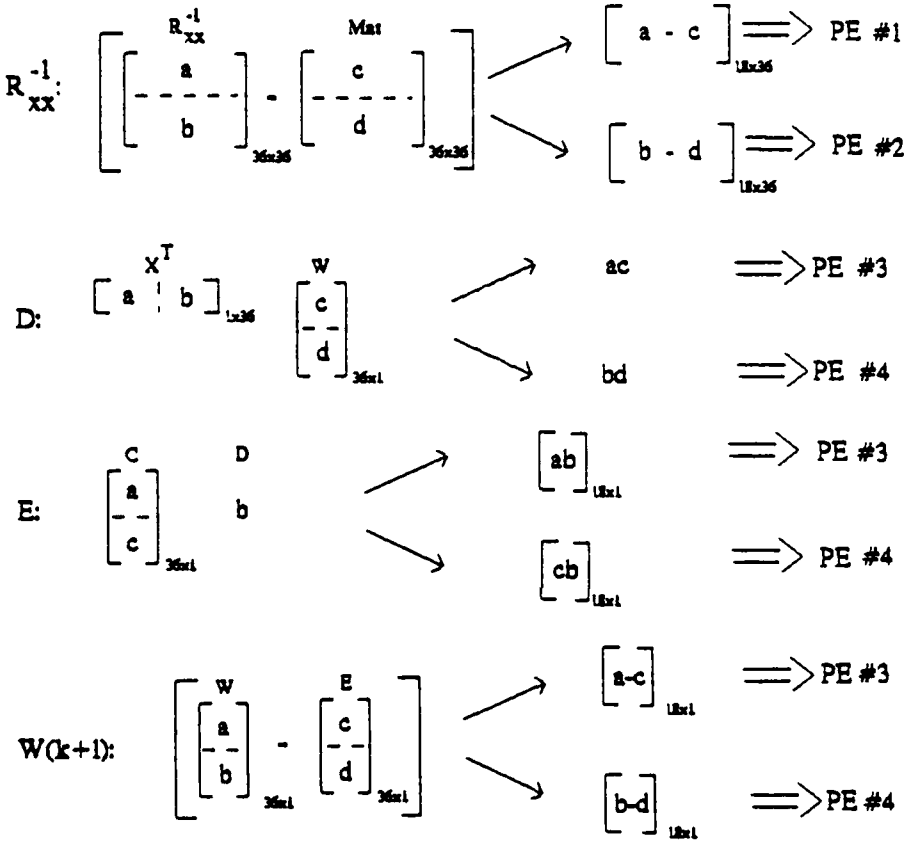


Figure 5.3 (cont.)



Now the code is written for each of the functions. The time taken by each function when operated in this environment is determined with the aid of the function time scheduling (see Figure 5.4). The iteration time obtained is 2.65 ms which is less than the iteration time of 3.8 ms imposed by the HF constraints. The time taken for each function in the system and the computational loading/chip for these functions are given in Table 5.4. Some results can be observed from this table. The time taken by each function is less than that obtained from execution time budgets analysis. This results because in the loading analysis, the functions required somewhere between one and two chips to compute. In the system described previously 2 chips were used for each function, thereby, as more computational power is available, the time taken by the functions are less than that obtained in the initial loading analysis.

The operation which was used to determine the speed of a DSP chip for the update covariance algorithm, was a 'complex multiply and add'. This provided a speed of 5.7 MOPS per chip. As functions A and B use this basic operation repeatedly, the DSP chips operate at near full speed while computing these functions. Mat function requires only complex multiply and involves no complex add, therefore the speed of the DSP chip while computing this function is 5.99 MOPS. Note that while computing Rxx the chips operates at 2.5 MOPS. The chips operates at this speed because this function requires isolated subtractions, and as in a DSP chip, a subtraction requires the same amount of time as a multiplication. This reduces the

Figure 5.4 Function scheduling diagram in the DSP architecture environment for the Update covariance algorithm

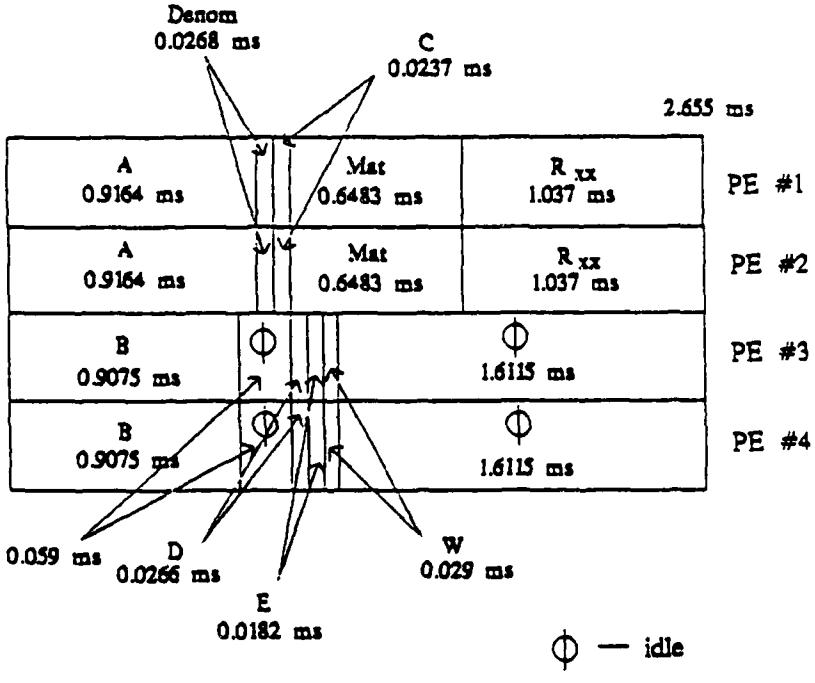


Table 5.4 Timing summary of DSP architecture implementation of Update covariance algorithm

Functions	Execution Time (ms)	Computational loading/chip
A	0.9164	5.63
Denom	0.0268	5.35
C	0.0237	4.767
B	0.9075	5.673
Mat	0.6483	5.99
Rxx	1.037	2.5
D	0.0266	5.375
E	0.0182	5.934
W	0.029	2.5

Table 5.5 Processor Utilization for the DSP architecture consideration of Update covariance algorithm

Processors	Processor utilization	Idle Time (ms)
#1 & #2	100 %	-
#3 & #4	37 %	1.674

performance of the DSP system. As the most computationally intensive functions, A and B are operated at near full speed, it can be said that the chips in the system are used efficiently. In fact we can determine the chip utilization [8] in the 4 chip system. Table 5.5 shows the utilization or fraction busy time and the idle time of each chip used. The chips #1 and #2 perform similar activities and are operational the whole iteration time. This resulted as chips #1 and #2 perform the functions A, Denom, C, Mat and Rxx which dictated the execution time budgets as discussed earlier. Therefore, chips #1 and #2 operated at 100 % utilization. Chips #3 and #4 perform functions B, D, E, and W which are in total less computationally intensive that the functions operated by chips #1 and #2. This can be noticed in Figure 5.4 where the chips #3 and #4 are idle 1.674 ms of total iteration time of 2.665 ms. This means chips #3 and #4 operate only at (2.655 - 1.674)/2.655 = 0.37 or 37 % utilization. The overall processor utilization of the 4 chip system is found to be

$$\frac{4(2.655) - 2(1.674)}{4(2.655)} = 68\%$$

The data memory needed by the update covariance algorithm when implemented in this system is 12.1 Kbytes (each complex word is represented as 16 bits real and 16 bits imaginary). Mat and Rxx each requires N^2 complex words of storage, X requires 144 complex words of storage (remember a copy of X is provided to each chip), X^* requires 72 complex

words, A, W and E each requires 36 complex words, C requires 72 complex words of storage and B requires 108 complex words of storage.

In summary, an evaluation of the update covariance algorithm using DSP architecture has been performed. The complexity of the algorithm was determined to be 4. A system architecture employing 4 DSP chips was designed and the resulting iteration time was 2.655 ms. The system performed 37 iterations during 100ms, ensuring on an average one convergence during this period. The data storage requirement was determined to be 12.1 Kbytes.

5.5 VLSI Architecture

This analysis starts with the identification of the compute bound operations. One of the compute bound operations to be implemented in a VLSI computing structure is a outer-vector product. The Wavefront architecture was found to be more suitable than the Systolic architecture for this outer-vector product. Next the Wavefront array design is presented for the compute-bound operations belonging to the update covariance algorithm. Finally the reasons for the selection of Wavefront architecture for the update covariance algorithm are presented.

5.5.1 Algorithm considerations

To aid in identifying the compute bound opertions, the algorithm is broken down into the following operations.

$$1.\ A = R_{-xx}^{-1}(k)\, X_{-}^{*}(k+1); \qquad B = X_{-}^{T}(k+1)\, R_{-xx}^{-1}(k)$$

Each function is a vector-matrix multiplication and the result is a vector. Each function needs $O(N^2)$ multiply-add steps and requires $O(N^2)$ input/output elements. When the order is the same, then we have to consider if the data once accessed from memory can be used many times in the computing structure. It can be seen for the A function that the elements of X^* once accessed can be used for each of the row of $Rxx^{-1}(k)$ and X^* multiplication. It can be seen for function B that the elements of X^T once accessed can be used for each of the column of $Rxx^{-1}(k)$ and X^T multiplication. Thus the functions A and B can be considered to be compute-bound and therefore can be implemented in VLSI computing structures.

$$2.\ Denom = X^{T}(k+1)\, A\ ; \qquad D = X^{T}(k+1)\, W(k)$$

Each function is a inner-vector product and the result is a scalar. Each function needs O(N) input-output elements and O(N) multiply-add steps. It can be seen that data once accessed from memory has no more further use. Therefore these functions are not suitable for VLSI computing structures.

3. C = A (1/Denom) ; E = C D

Each function is a scalar-vector product and the result is a vector. Each function requires N multiply steps and 2N+1 input-output elements. Therefore these functions are not suitable for VLSI computing structures.

5. Mat = C B

This function is a outer-vector product and the result is a matrix. This function needs $O(N^2)$ input-output elements and $O(N^2)$ multiply steps. This function is compute bound as both the vectors once accessed can be reused until the computations are completed.

6. $R'xx = [Rxx(k) - Mat]^{-1}$

This function is matrix-matrix subtraction and requires $3N^2$ input-output elements and N^2 subtractions. This means that this function is input-output bound and is not suitable for VLSI computing structures.

7. $W' = [W(k) - E]$

This function is a vector-vector subtraction and requires 3N input-output elements and N subtract steps. This function is a input-output bound operation.

The discussion performed is summarized in Table 5.6. Thus the VLSI computing structures for the compute bound operations A, B and Mat are designed in the next section.

5.5.2 Wavefront Array Design

As previously determined the Wavefront architecture [10] was chosen for the Update covariance algorithm. The important Wavefront array system parameters to be determined in this section are:

1) arrangements of the PE's to compute each compute-bound operation
2) computational capacity of the PE

Table 5.6 List of compute-bound and Input-output bound functions belonging to Update covariance algorithm

Functions	Type of operation	Number of computations	Number of input-output elements	Operation bound
A	vector-matrix product	$O(N^2)$	$O(N^2)$	compute
Denom′	inner vector product	O(N)	O(N)	input-output
C	scalar-vector product	N	2N+1	input-output
B	vector-matrix product	$O(N^2)$	$O(N^2)$	compute
Mat	outer vector product	$O(N^2)$	$O(N^2)$	compute
Rxx′	matrix-matrix subtraction	N^2	$3N^2$	input-output
D	inner vector product	O(N)	O(N)	input-output
E	scalar-vector product	N	2N+1	input-output
W′	vector-vector subtraction	N	3N	input-output

3) bandwidth of the Wavefront array bus

4) host computational capacity

5) buffer size

The characteristics of the PE to be used in the Wavefront array are analyzed now. The PE has an architecture consisting of an internal program memory, a control unit, an ALU, and a set of registers [10]. Time must be determined by the various instructions performed by the PE. For the update covariance algorithm, the PE has very few instructions to perform - multiply, add, fetch, flow and transfer (register to register). The fetch operation fetches a data element from a neighboring PE or from a memory module via handshaking procedures. The flow operation sends a data element from a PE to either its neighboring PE or the memory module. The clock period for the PE is chosen to be 30ns i.e., T(pe) = 30ns. The allocation of the number of cycles needed for the different instructions performed by the PE, is determined by the number of cycles taken to perform similar instructions by the 32-bit microprocessor Mc 68020, and are:

Instructions	cycles	time(μs)
Multiply	28	0.84
Add	7	0.21
Transfer	3	0.09
Fetch/Flow	4	0.12

The Wavefront array design to compute the compute bound functions A, B and Mat is presented next.

5.5.2.1 Wavefront array design for function 'A'

The arrangement of the PE's performing the function A is

shown in Figure 5.5. A linear array of 36 PE's is needed to compute the function A when the number of antenna elements N = 36. The matrix-vector product can be decomposed into 36 vector-vector products; each vector-vector product is the multiplication of a row of the matrix Rxx and the vector X^*. There are 36 PE's and each PE is responsible for a vector-vector product. Each vector-vector product constitutes 36 complex multiplication and addition. The matrix Rxx is broken down into rows and one row is provided to each PE. The X^* vector is provided to the first PE only and the elements of this vector are passed from PE to PE in a pipelined fashion. The capability of reusing this vector X^*(the vector is accessed from memory only once) is the reason for the speedup (when compared to operation of function A on a uniprocessor) achieved by the VLSI structure. Each element of the vectors is a complex quantity. The real and imaginary components are represented by 16 bits. When an element is supplied to a PE the real component is provided first followed by the imaginary component. These elements are stored in memory modules and are provided to the PE whenever the PE requires them.

Remember that Wavefront array is an asynchronous system where a PE obtains the operands from neighboring PE's through handshaking procedures. Also the Wavefront arrays behave like a data flow machine in that once the operands are fetched by a PE, it at once begins its operations on the data obtained. The operations to be performed by each PE, to compute function A, is coded in Matrix Data Flow Language (MDFL) (see Figure 5.6).

Figure 5.5 Wavefront array to compute function 'A'

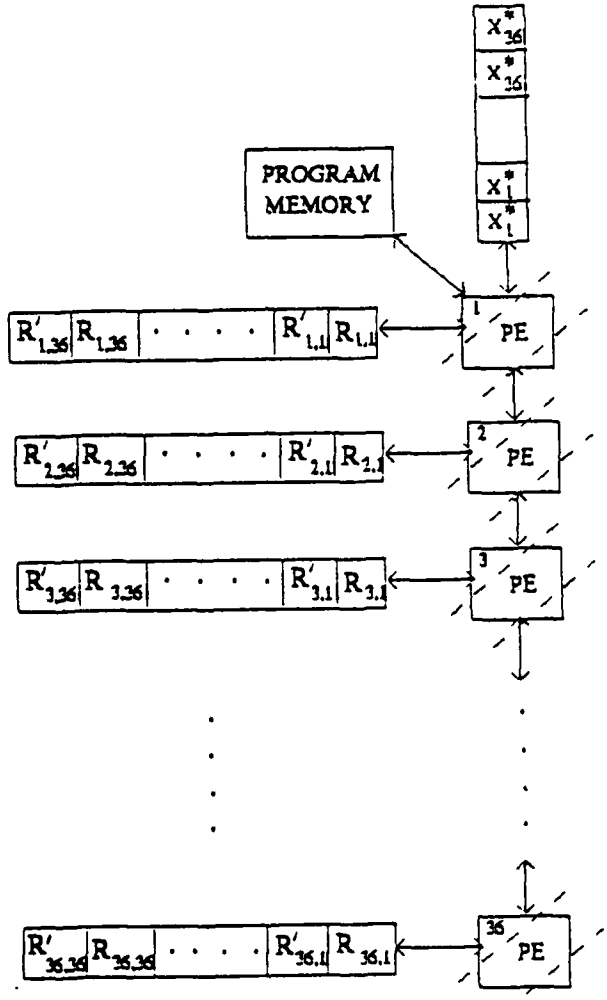


Figure 5.6 Matrix data flow language program for function A

```
BEGIN
 SET COUNT 36;
  REPEAT
    WHILE WAVEFRONT IN ARRAY DO
     BEGIN
      FETCH B UP;
      FETCH A LEFT;
      FLOW B DOWN;
      TSR A,R1;          {TSR implies transfer}
      TSR B,R2;
      MULT A,B,R3;
     END;

    WHILE WAVEFRONT IN ARRAY DO
     BEGIN
      FETCH B UP;
      FETCH A LEFT;
      FLOW B DOWN;
      MULT A,B,R4;
      SUB R3,R4,R3;
      ADD C,R3,C;      { real part in C register}
      MULT R2,B,R2;
      ADD R1,R2,R1;
      ADD C1,R1,C1;    { imaginary part in C1 register}
     END;
 DECREMENT COUNT;
UNTIL TERMINATED;

BEGIN
SET COUNT 1;
REPEAT
 WHILE WAVEFRONT IN ARRAY DO
  BEGIN
   FETCH A UP;
   FLOW C LEFT;
   FLOW C1 RIGHT;
   FLOW A DOWN;
  END;
 DECREMENT COUNT;
UNTIL TERMINATED;
END PROGRAM.
```

Notice that 36 recursions are involved. Each recursion consists of two Wavefronts which pass through each PE. Each PE is responsible to complete 36 complex multiplication and addition. In every recursion, a PE computes one complex multiplication and addition. The registers in the PE are first initialized to zero. During the first Wavefront of a recursion, a PE 1) fetches the two real components, one belonging to matrix Rxx and the other belonging to vector X^*, 2) sends the real component of X^* to the next PE in the pipeline, 3) stores the two operands in temporary registers and 4) performs a multiplication of the two operands. A complex multiply-add requires 4 real multiplies and 4 real adds/subtracts. During the second Wavefront of the recursion the PE then performs the remaining 3 real multiplies and 4 real adds/subtracts when it fetches the imaginary components of the matrix Rxx and vector X^*.

The time taken by the Wavefront array to perform the function A can be now determined. The first Wavefront of the recursion requires 46 cycles and is found to be

instructions	number of cycles
2 fetches	8
1 flow	4
2 transfer	6
1 multiply	28
	46

The time taken by the PE to perform the first Wavefront is 46 * T(pe) = 46 * 30ns = 1.38 μs. The time taken to perform the second Wavefront of the recursion is found to be

instructions	number of cycles
2 fetches	8
1 flow	4
3 multiplies	84
4 adds	28
	124

The time taken by the PE to perform the second Wavefront is 124 * 30ns = 3.72 μs. Thus the total time required by a PE to perform one recursion (involves one complex multiplication and addition) is 5.1μs. Each PE then has a computational capacity of 1.6 MOPS (performs 8 operations in 5.1 μs). In general, if a recursion requires K{mult}, K{add}, K{fetch}, K{flow} and K{transfer} cycles, the the PE would require an execution time of:

[K{mult} + K{add} + K{flow} + K{fetch} + K{transfer}] * T(pe).

At the end of 36 recursions the results of the 36 complex multiplication and addition performed by each PE resides in the registers C and C1 (see Figure 5.6). The 36 complex quantities residing in the 36 PE's are the resultant vector. This resultant vector is sent to the buffer so that the host can access it to perform further functions of the update covariance algorithm. This caused one more recursion to be passed through each PE as shown in Figure 5.6. During this Wavefront, each PE flows the contents of the registers C and C1 to the buffer. This Wavefront needs 16 cycles which requires 8 * T(pe) = 16 * 30 = 0.48 μs. A dummy fetch is included, since one of the requirements for the Wavefront language is that every Wavefront computation should be preceded by a fetch [10].

Once the time taken to perform a recursion is known, time required by the Wavefront array to complete the computations required by the function A can be figured. The reason for breaking the recursion into two Wavefronts becomes apparent. These two Wavefronts introduce one more level of pipelining which increase the speedup factor. When the second Wavefront of the first recursion is operating on the first PE of the array, the first Wavefront of the same recursion will be operating on the second PE. The first PE takes 36 * (execution time for one recursion) to complete the 36 complex multiplication and addition assigned to it. Once the first PE finishes its computations, the remaining 35 PE's will finish their computations when the second Wavefront of the 36th recursion passes through the 35 PE's. Notice that the second Wavefront takes more time than the first Wavefront. At this instant, the 36 recursions are performed by each PE and the results are in the registers of each PE. One more recursion is needed to send the data out and this requires 36 Wavefronts. Thus the total time required to perform the function A is

= 36 (execution time/recursion) + 35 (execution time for the second Wavefront) + 36 (execution time required to send the data out)

= 36 (5.1) + 35(3.72) + 36(0.48) = 331.08 μs .

In general the above expression can be written for any number of antenna elements, N, as

= N(execution time/recursion) + (N-1) (execution time for second Wavefront) + N (execution time required to send the data out).

If the recursion was not split into two Wavefronts, it can be seen that the time required to compute function A is (36 * 5.1) + (35 * 5.1) + (36 * 0.48) = 379.38 μs . By introducing one more level of pipelining by dividing the recursion into two Wavefront, as performed in our analysis, the time is reduced for the execution of function A.

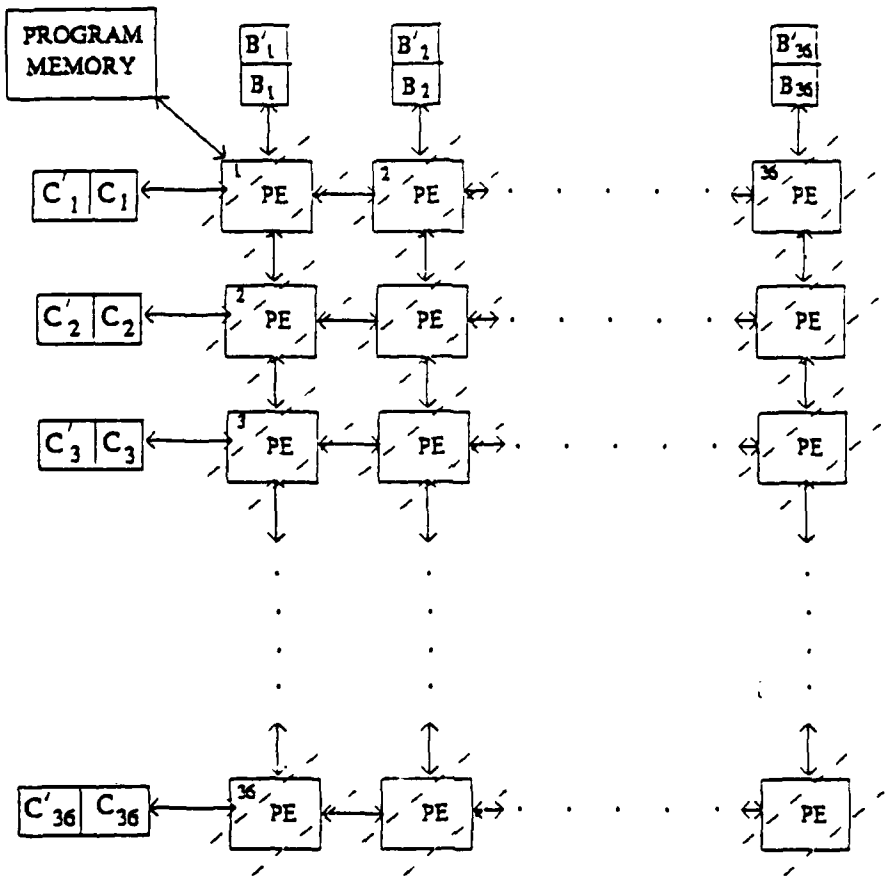
5.5.2.2 Wavefront array design for function 'B'

Function B is similar to function A as it also involves a matrix vector product. A linear array of 36 PE's is required for performing 36 complex multiplications and additions. Each column of matrix Rxx is provided to each PE and the vector X^T is provided to the first PE only. The procedure in determining the time taken to compute function B is similar to that performed for function A, thus the time taken by the Wavefront array to compute function B is 331.08 μs .

5.5.2.3 Wavefront array design for function 'Mat'

The Mat function is a outer-vector product resulting in a matrix. A square array of 36 * 36 PE's is required (for N = 36) as shown in Figure 5.7. The 36 elements of the C vector are sent to the 36 PE's belonging to the first row of the square array (i.e., one element/PE). Similarly the 36 elements of B vector are sent to the 36 PE's belonging to the first column of the square array (i.e., one element/PE). Notice that the

Figure 5.7 Wavefront array to compute function 'Mat'



elements of the vector B and C are accessed only once for the computation of Mat in this array. This increases the speedup, when compared to this function operated on a uniprocessor.

The code written in MDFL and stored in each PE to perform the Mat operation is shown in Figure 5.8. The Mat function requires 36 * 36 complex multiplications and as there are 1296 PE's each PE has the task of computing one complex multiplication. This means that one recursion is sufficient for this purpose. (see Figure 5.8) The recursion is broken down into two Wavefronts. During the first Wavefront, the PE fetches the two real components, stores them, flows the two real components to neighboring PE's (down and right) and performs a real multiplication. During the second Wavefront, the PE performs the remaining 3 real multiplies and 2 real adds/subtracts to complete the complex multiplication.

The time taken by a recursion can be determined as follows. The first Wavefront requires 50 cycles as shown.

instructions	cycles
2 fetches	8
2 flows	8
2 transfer	6
1 multiply	28
	50

Thus the first Wavefront requires 50 * T(pe)= 50 * 30ns= 1.5 μs The second Wavefront requires 114 cycles as shown.

instructions	cycles
2 fetches	8
2 flows	8
3 multiplies	84
2 add/subtract	14

Figure 5.8 Matrix data flow language program for Mat function

```
BEGIN
 SET COUNT 1;
  REPEAT;
    WHILE WAVEFRONT IN ARRAY DO
     BEGIN
      FETCH B UP;
      FETCH A LEFT;
      FLOW A RIGHT;
      FLOW B DOWN;
      TSR A,R1;
      TSR B,R2;
      MULT A,B,R3;
     END;

    WHILE WAVEFRONT IN ARRAY DO
     BEGIN
      FETCH B UP;
      FETCH A LEFT;
      FLOW A RIGHT;
      FLOW B DOWN;
      MULT A,B,R4;
      SUB R3,R4,C;
      MULT R1,A,R1;
      MULT R2,B,R2;
      ADD R1,R2,C1;
     END;
  DECREMENT COUNT;
 UNTIL TERMINATED;

BEGIN
 SET COUNT 36;
  REPEAT
    WHILE WAVEFRONT IN ARRAY DO
     BEGIN
      FETCH A RIGHT;
      FETCH B DOWN;
      FLOW C LEFT;
      FLOW C1 UP;
      TSR A, C;
      TSR B, C1;
     END;
   DECREMENT COUNT
  UNTIL TERMINATED
END PROGRAM.
```

Thus the second Wavefront requires 114 * 30ns = 3.42 μs.

At the end of this recursion, one complex multiplication has been completed by all the PE's; the result of this complex multiplication is stored in the registers C (real) and C1 (imaginary). This data (the result) is shifted and brought outside the Wavefront array. This required N recursions to be passed through the array which required (2N-1) time units. During each Wavefront, the PE gets the contents of register C from the cell to its right and the contents of register C1 from the cell below and stores them. It also sends the contents of its registers to the left PE and the PE above. The real components of the Mat matrix then comes from the 36 PE's of the first column and the imaginary components come from the 36 Pe's of the first row. This Wavefront requires 22 cycles as shown.

instructions	cycles
2 fetches	8
2 flows	8
2 transfers	6
	22

This Wavefront requires 22 * T(pe) = 22 * 30ns = 0.66 μs. The total time required by this Wavefront array to compute Mat function is as follows.

Total time = (execution time/recursion) + 70 * (execution time for second Wavefront) + 71 * (execution time/Wavefront to send the data out)
= (1.5 + 3.42) + 70 * (3.42) + 71 * (0.66)
= 291.18 μs.

In general for any N the total time is given as

$$= (1.5 + 3.42) + (2N-2) * (3.42) + (2N-1)*(0.66)$$

5.5.3 System parameters

The Wavefront array is interfaced to a host [9],[12]. The parameters of this array processor system will be discussed.

5.5.3.1 Host

The host performs the input-bound operations belonging to the Update covariance algorithm. The input-output bound functions require 6338 operations as shown.

input-output bound functions	operations
Denom	287
C	221
Rxx	5184
E	216
W	144
D	286
	6338

Compute bound functions A and B are computed in parallel and require 331.08 μs, Mat function requires 291.18 μs, thus the total time required by the compute bound operations is 622 μs. The iteration time for the Update covariance algorithm is 3.8 ms. The input-output operations must be performed in 3.16 ms which then requires a host of computational capacity of 2 MOPS. Remember, as the host is interfaced with the high speed array processor units, a host is chosen not only with a computational capacity of 2.0 MOPS, but also with a host bus that does not form a bottleneck while transferring data to and from the Wavefront arrays.

5.5.3.2 Wavefront array bus bandwidth

Three Wavefront arrays were selected to find the compute bound functions. These Wavefront arrays were then connected to a Wavefront bus which carries the input and output data for the Wavefront arrays. The Wavefront arrays consume data at a very high rate which results in a large bandwidth for the Wavefront bus. The number of words consumed and supplied by an array is easier to be determined for the Systolic array than for the Wavefront array. This results because in Systolic arrays each PE consumes data with respect to a time reference, whereas in Wavefront array each PE consumes data whenever the data is available. Here, the worst case situation is determined. During the worst case, at the peak instant, the linear array computing function A consumes 37 words : the 36 PE's consume 36 words of Rxx and the first PE consumes one word of X^*. As PE's require 4 cycles or 120 ns to fetch a word, the Wavefront needs 37*2=74 bytes to be fetched in 120ns. The Wavefront array needs 616 Mbytes/sec bus bandwidth. Function B operates similarly to function A and it also needs 616 Mbytes/sec bus bandwidth.

The Mat function consumes only 2 words per Wavefront, but while the data stored in the registers of the PE's are brought out, the Mat function outputs at the peak instant (worst case) 72 bytes in 120 ns (36 words out of first row and 36 words out of first column). The Wavefront array computing Mat function requires 600 Mbytes/sec.

As the Wavefront arrays computing function A and B operate in parallel, the bandwidth required of the bus is doubled to

1232 Mbytes/sec. The Mat function is computed separately and requires only 600 Mbytes/sec to fix the bandwidth of the Wavefront array bus to be 1232 Mbytes/sec.

5.5.3.3 Buffer

The buffer is required to store the data provided to the Wavefront arrays and the data received from the Wavefront arrays. The buffer has to allocate N complex words each for X^T, X^*, A, B and C. The buffer also has to allocate N^2 complex words for Rxx and Mat. In total the buffer requires 11088 bytes (for N=36) where each complex word requires 4 bytes.

A summary of the various parameters discussed for the Wavefront architecture is shown in Table 5.7.

5.5.4 Architectural considerations

Justification will be given for choosing Wavefront architecture over the Systolic architecture for the update covariance algorithm. It will be shown that the Mat operation (outer-vector) product was the reason to choose Wavefront architecture. The issues considered in deciding on the suitable architecture follows.

5.5.4.1 Speed Variation

A Wavefront array enjoys the performance advantage that results from each PE of the array being able to start computing as soon as its inputs are ready and to make its output available as soon as it is finished computing. If different

Table 5.7 Summary of VLSI architecture parameters for Update covariance algorithm

Parameters	Functions		
	A	B	Mat
Number of PE's	36	36	1296
computational capacity/PE	1.6 MOPS	1.6 MOPS	1.6 MOPS
time/iteration	331 μs	331 μs	291 μs
speedup factor	18.5	18.5	18.5
wavefront array Bandwidth(peak)	616 Mb/sec	616 Mb/sec	600 Mb/sec

Array Processor System Requirements

Host computational capacity	2.16 MOPS
Bandwidth of wavefront array bus	1232 Mb/sec
Buffer memory required	11088 bytes

kinds of computations are taking place in the computing array, i.e., speed variations in computations exist, then the data-dependence property of Wavefront arrays will be an advantage. This advantage is used in the Update covariance algorithm when two different computation (Wavefronts) were pipelined for every recursion. These two computations require different speeds (need different execution times) and if operated on a Systolic array, then the array will be clocked at the rate of the slowest operation. Let us compare the time taken for the 36^2 complex multiplications performed for the Mat operation by the Systolic and the Wavefront arrays. Note that in the asynchronous scheme, (Wavefront) flow and fetch are separate tasks; in a synchronous array (Systolic) with a two phase clock, they may be combined into one task [11] that involves a simultaneous flow to left and fetch from right, or flow down and fetch from up. This combination is clearly an attribute and is taken into consideration in the synchronous (Systolic) model. When the Mat operation takes place in the Systolic array, there exists no speed variation and in each time unit the PE's perform a complex multiplication. The Systolic array requires (2 N -1) * (execution time/complex multiplication) = (71 * (1.26 + 3.18)) = 315.24 μs. The time of 315.24 μs is for the 36 complex multiplications and the result is still in the PE registers. In the Wavefront this 36 complex multiplications due to speed variations require only 244.32 μs a savings of 71 μs over the Systolic implementation. Also note that though handshaking procedures [11] are used in exchanging

data between PE's, no time is lost in doing so. This results because after a 'flow' has been executed by a PE, it starts implementing its next task concurrently with the 'fetch' executed by the neighboring PE, hence no waiting.

Thus from the speed variation issue the Wavefront array is more suited than the Systolic array for the Mat operation.

5.5.4.2 Clock skew

The Systolic array is an example of a totally synchronous system and therefore suffers from clock skew. The clock skew phenomenon arises from three factors [11]

1. the RC of the global distribution network
2. the variance in values of gate threshold voltage (Vt) of the PE gate
3. unequal clock paths to various PE's in the array

When different PE's receive clock signals by different paths, they may not receive clocking events at the same time, potentially causing synchronization failure. To eliminate the clock skew due to unequal clock paths, the PE's of the square array use a H-tree clock distribution network [11] (Figure 5.9). If the H-tree is complete, then the clock skew is due to the R,C and Vt and it increases as $O(N^3)$; where N is given by the relation

$$q = \log_2 N$$

and where q is the number of levels of the network. The Mat operation employs an array of 1296 PE's (36 by 36). Now for a square array of 1296 PE's, there are 6 levels, therefore, the

Figure 5.9 H-tree clock distribution network for square arrays [11]

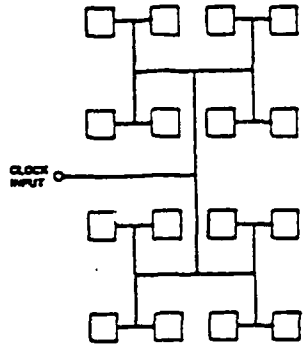
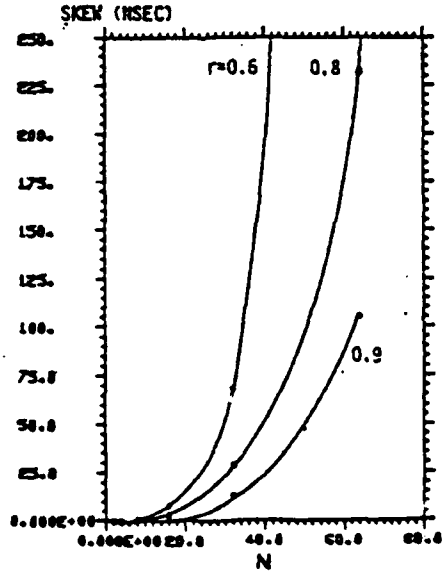


Figure 5.10 Clock skew vs N [11]



clock skew increases as O(64**3). It has been determined by simulation [11] that the clock skew is a function of N and is shown (for different values of metal content (r) of clock distribution path) in Figure 5.10. It can be seen for N=64 the clock skew is about 100 ns. The clock skew in the 1296 PE array will be higher than this as the H-tree is not complete (needs 4096 PE's for a complete H-tree for level 6). In the best case, the PE in the Systolic array will be clocking at clock period of T(ck), where T(ck) - T(pe) = T(skew). The clock period in the array is T(ck) = 30 + 100 = 130ns which increases the time required to compute the 36 complex multiplication needed, from 315.24 μs to 1.36 ms.

One way the clock skew can be reduced is by partitioning the problem and using a smaller array. For example, if a square array of 18 by 18 PE's is used, then the clock skew is reduced considerably. But the price paid is in increased time for Mat operation computation; the speed up has therefore been reduced by 0.75. This is easily seen as the square array has to be traversed four times to complete the 36 complex multiplication on a 18 by 18 square array. Remember that the time taken to get the results stored in the registers of PE's out of the array will increase. Also the PE's will require more registers as 4 complex words of the result are stored.

Due to the noted results, the Wavefront array was considered more favorable than the Systolic array for the Mat operation; hence the Update covariance algorithm.

The Wavefront design has interrelated system parameters. If

the bandwidth of the Wavefront bus needs to be lowered, a slower PE is needed. This increases the computation time of the compute bound operations which, in turn, leads to a need for a host with a computational capacity of more than 2 MOPS.

5.6 Chapter summary

In this chapter hardware evaluations were performed to determine the suitability of the Update covariance algorithm for general purpose microprocessor architecture, Digital signal processor architecture and the VLSI architecture.

The following isssues were considered while evaluating the various architectures:

1) the computational complexity of the Update covariance
2) the input requirements - the input samples arrive every 3.8 ms
3) the characteristics of the architecture

Considering these points, complexity was obtained for each architecture. A complexity of 27 for the microprocessor architecture and 4 for the DSP architecture was found. More chips were needed by the microprocessor architecture because the microprocessor architecture needs 16 μs to compute a complex multiply-add, and the DSP architecture requires only 1.4 μs. Next an architecture was developed for the update covariance algorithm in a 4 DSP chip environment. It was found that the iteration took 2.655 ms to provide 37 iterations during the period of 100ms. This ensured, on a average, at least one convergence per 100ms. Due to the requirement of

storage of intermediate results, this environment required 12.1 Kbytes of data storage.

Next, two VLSI architectures, Systolic and Wavefront, were considered for the Update covariance algorithm. One of the compute bound operations was an outer vector product, demanding a square array of 1296 PE's (using 36 antenna elements). Due to clock skew and speed variation factors, the Wavefront array was considered a better suited architecture than the Systolic for the outer vector product, hence, the Update covariance algorithm. The number of PE's that could be placed on a single chip is limited by the present technology. A PE with a clock cycle of 30 ns was used which established the requirement of 0.622 μs to compute the compute bound operations. This result meant that the Wavefront bus had to have a bandwidth of 1232 Mbytes/sec, and a host computational capacity of 2 MOPS. The result is a very high complexity for the VLSI architecture design.

It can be firmly concluded from the analysis, the DSP architecture is best suited for the update covariance algorithm.

6.0 CONCLUSIONS AND RECOMMENDATIONS

A feasibility study was performed to determine the suitability of the HF adaptive control algorithms for the general purpose microprocessor architecture, the Digital signal processor architecture and the VLSI architecture. The complexity of the adaptive algorithms considered for study of the various hardware architectures is indicated in Table 6.1. The digital signal processor architecture has been shown to be the best suited architecture for the adaptive algorithms that were considered. The reason being that the sum-of-products computations dominated in the algorithms considered for study, and arithmetic units, performing the basic function AX + Y -> Y are best suited, such as present in DSP architecture.

For adaptive algorithms, feedback exists from the output back to the input that requires the total input-to-output delay be less than one sample period. This total latency constraint (determined by previous simulation studies) causes a cascaded (pipelined) chain of processors to be unsuitable for adaptive algorithms. System architectures were developed for the feasible hardware structures and the analysis indicates that a parallel configuration (in contrast to cascade configuration) allows multiprocessing by providing the output within one sample period. For the adaptive algorithms considered the time/iteration achieved by the best suited architecture is shown in Table 6.2. The reason for developing system architectures for the best suited technology is to determine whether the time involved in the communication of data between

Table 6.1 Complexity of the adaptive algorithms for the various hardware architectures considered

Algorithm	Architectures		
	microprocessor	DSP	VLSI
Lms	12	1	-not applicable-
c-Lms	281	23	6 - 9 (custom chips) + 10 MOPS (host)
Update covariance	27	4	- high -

Table 6.2 Time/iteration achieved by the system architectures developed for the best suited architectures

Algorithm	Initial assessment on the number of DSP chips	Time/iteration HF constraints	Number of DSP chips used for system design	Time/iteration system design	Data memory requirement (words)
LMS	1	107.5 μs	1	105.8 μs	147
c-LMS	23	86 μs	26	83.9 μs	2808
update	4	3.8 ms	4	2.6 ms	6192

processors introduces additional computational capacity for the algorithms that further increases the complexity. Note that the LMS algorithm requires a loosely coupled system as the update of one weight is independent of the update of another, whereas the c-LMS algorithm results in a tightly coupled system because of large communication overhead due to passing of intermediate data.

The c-LMS algorithm was the most complex algorithm to implement. Though both the update covariance and the c-LMS algorithm have the computational complexity of $O(N^2)$; due to the presence of parallelism among functions and the requirement of fewer iterations to converge (by the update covariance algorithm) makes the c-LMS algorithm more complex. The DSP architecture was a better choice because of the nature of the complex interface between the host and the Systolic array, and the requirement of large number of pins needed by the chips in the Systolic array. The VLSI computing structure is a better choice as the number of antenna elements in the adaptive array increase. For example, when the number of antenna elements increases to 64, 70 DSP chips are required to compute the bottleneck operation, matrix-vector product, to obtain an output every sampling period. The Systolic array requires only between 12 to 16 custom VLSI chips, each chip performing simple operations.

Linear code was used whenever possible in the system architecture to increase speed. A fixed point implementation (as time per recursion is small) providing 16 bits each for the

I and Q channels (dynamic range of 96 db). In this study fixed point implementation with truncation was used. This results in numerical problems [28]. With the advent of faster floating-point DSP chips [29], and to avoid numerical problems the floating-point DSP processors is recommended.

REFERENCES

[1] Darin S. Haegert, "Adaptive Algorithms for HF Antenna Arrays," M.S. project report, Dept. of Electrical and Computer Engineering, 1984.

[2] "MC68020 32-bit Microprocessor User's Manual," Prentice Hall, Inc., Englewood Cliffs, N.J, 1984.

[3] M. Schwartz, J. Schiappacasse, and G. Baskerville, "Signal Processor's Multiple Memory Buses Shuttle Data Swiftly," Electronic Design, Feb 20, 1986.

[4] H. Ahmed, J. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," IEEE Computer, Jan 1982.

[5] H. Whitehouse, J. Speiser, and K. Bromley, "Signal Processing Applications of Concurrent Array Processor Technology," VLSI and Modern Signal Processing, S.Y. Kung et al. eds., Prentice-Hall, Inc., Englewood Cliffs, 1985.

[6] H.T. Kung, "Why Systolic Architectures," IEEE Computer, Jan 1982.

[7] C. Mead, and L. Conway, Introduction to VLSI Systems, Addison-Wesley publishing Company, Chapter on 'Algorithms for VLSI Processor Arrays', by H.T Kung, and C.E. Leiserson.

[8] K. Hwang, and F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill Book Company, 1984.

[9] S.Y. Kung, "VLSI Array Processors," IEEE, ASSP Magazine, July 1985.

[10] S.Y. Kung, K.S. Arun, R.J. Gal-ezer, and D.V. Bhaskar Rao, "Wavefront Array Processor: Language, Architecture, and Applications," IEEE Transactions on Computers, Nov 1982.

[11] S.Y. Kung, and R.J. Gal-ezer, "Synchronous versus Asynchronous Computation in Very Large Scale Integrated (VLSI) Array Processors," SPIE Vol 341, Real time signal processing V, 1982.

[12] A.L. Fisher, and H.T. Kung, "Special-Purpose VLSI

Architectures: General Discussions and a Case Study," VLSI and Modern Signal Processing, S.Y. Kung et al., Editors. Prentice-Hall Inc., 1985.

[13] B. Widrow, P. Mantey, L. Griffiths, and B. Goode, "Adaptive Antenna Systems," Proceedings of the IEEE, Vol. 55, Dec. 1967.

[14] B.A. Bowen and W.R. Brown, Systems Design- Volume II of VLSI Systems Design for Digital Signal Processing, Prentice-Hall, Inc., 1985.

[15] V.B. Lawrence, and S.K. Tewksbury, "Multiprocessor Implementation of Adapative Digital Filters," IEEE Transactions on Communications, June 1983.

[16] B. Widrow, S.D. Stearns, Adaptive Signal Processing, Prentice-Hall, Inc., 1985.

[17] R.T. Compton Jr., "An Adaptive Array in a Spread Spectrum Communication System," Proceedings of the IEEE, March 1978.

[18] O.L. Frost III, "An Algorithm for Linearly Constrained Adaptive Array Processors," Proceedings of the IEEE, August. 1972.

[19] K. Takao, M. Fujita, and T. Nishi, "An Adaptive Antenna Array Under Directional Constraint," IEEE Transactions on Antennas and Propagation, Sept 1976.

[20] A.L. Fisher, and H.T. Kung, "Synchronizing large VLSI processor arrays," ACM, 1983.

[21] R. Schreiber, and P.J. Kuekes, "Systolic Linear Algebra Machines in Digital Signal Processing," VLSI and Modern Signal Processing, S.Y. Kung et al., eds. Prentice-Hall, Inc., 1985.

[22] E.E. Swartzlande, Jr., VLSI Signal Processing Systems, Kluwer Academic Publishers, 1986.

[23] A.V. Kulkarni, and D.W.L. Yen, "Systolic Processing and an Implementation for Signal and Image Processing," IEEE Transactions on Computers, Vol. C-31, No. 10, Oct 1982.

[24] J. Blackmer, G. Frank, and P. Kuekes, "A 200 Million Operations per Second (MOPS) Systolic Processor," SPIE Vol. 298, Real Time Signal Processing IV, 1981.

[25] R.A. Monzingo, and T.W. Miller, Introduction to Adaptive Arrays, John Wiley and Sons, N.Y., 1980.

[26] S.Y.Kung, "VLSI Array Processor for Signal Processing," presented at the Conf. Advanced Res. in Integrated circuits, M.I.T., Cambridge, Jan 28-30, 1980

[27] J.M.Speiser and H.J. Whitehouse, "Architectures for Real Time Matrix Operations," in Proc. GOMAC, Nov. 1980.

[28] T.Thong and B.Liu, "Fixed-pt Fast Fourier Transform Error Analysis," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-24, pp. 563-573, Dec. 1976.

[29] Amnon Aliphas and Joel A.Feldman, "The Versatility of Digital Signal Processing Chips," IEEE Spectrum, June 1987.